INDEX

1. Reversing DUY DaD Tape - Additional Protection Removal
2. Reversing Waves Plugins V7r12
3. Concepts for Writing an Emulator
4. Titan 4.0.6 Reversing Notes
5. Edi Prompt 1.3.0 Crack Notes
6. Sysenter break and reversing in Safari
7. Reversing Unity 3.0 and additional Pace reversing
8. Reversing Protools 9.0 HD
9. General Hasp HL reversing Notes
10. Reversing Wave Rider 2

Reversing DUY DaD Tape - Additional Protection Removal

By unsanity (MSJ)

Step 1: Reconnaissance

or, ***What the Hell Am I Doing?***

Additional tools required:

a brain

Once you have the DUY DaD Tape AU plugin downloaded, go ahead and install it. These installers are actually old CFM files (they're not actually Mach-O files at all) so I got a little lazy in figuring out how to bypass the serial protection and instead posted the serials for the installers. The only files that are installed are the plugins themselves.

Now we'll run unpace on the plugin, and use just the -f option. (some people may have to use sudo)

Code:

```
unpace -f "/Library/Audio/Plug-Ins/Components/DaD Tape AU.component"
```

On this particular plugin (and some other AU plugins) we don't get a window that pops up with the quit option, but the feedback from unpace shows a successful unpack.

Now it's time to get a feel for what unpace has done. Go ahead and run your AU Host and see if it picks up the DUY DaD plugin.

Whoops! We get the trial window and once we press quit, we get this nice little message.

DaD Tape native could not be initialized due to the following error:
(Reference: -7300)
This problem may be due to the software authorization. For further information on registering and authorizing DUY software, please contact info@duy.com, quoting the error reference above.
We also remind you that full details on authorizing can be found in the Installation Guide included with this product.
OK

OK, so the plugin does not crash, but it looks as if the PACE protection is still there! Well, here is where we use the trick of breaking on entrypoint. So we'll use GDB to find out where PACE is getting called from.
The next step I like to do because my .gdbinit file is 32-bit only, is separate out the i386 architecture from AU Lab. This way, even on a 64-bit capable machine, only the 32-bit code is ran. So I do it like this:

Code:

```
cd /Developer/Applications/Audio/AU\ Lab.app/Contents/MacOS/
lipo ./AU\ Lab -extract i386 -output ./i386_AU_Lab
```

and then we'll go ahead and run it under GDB:

Code:

```
gdb /Developer/Applications/Audio/AU\ Lab.app/Contents/MacOS/i386_AU_Lab
```

If you do not have a gdbinit file, I HIGHLY recommend you use one. I'll be posting the one I use at the end of this post. Mine has been modified from fG!'s (reverse.put.as) to include the frame as well as the stack.
So once GDB has loaded, we'll go ahead and set a future breakpoint on IL_SC_D_PPC. If you use my gdbinit file, or some other ones found on the internet, breakpoint pending will already be set to on. If not, first thing is to go ahead and turn it on, then source your gdbinit file. I do not place my gdbinit file in ~/.gdbinit because doing that messes up using the debugger in XCode. So i place it in the root of my drive, and just source it everytime I start GDB.

Code:

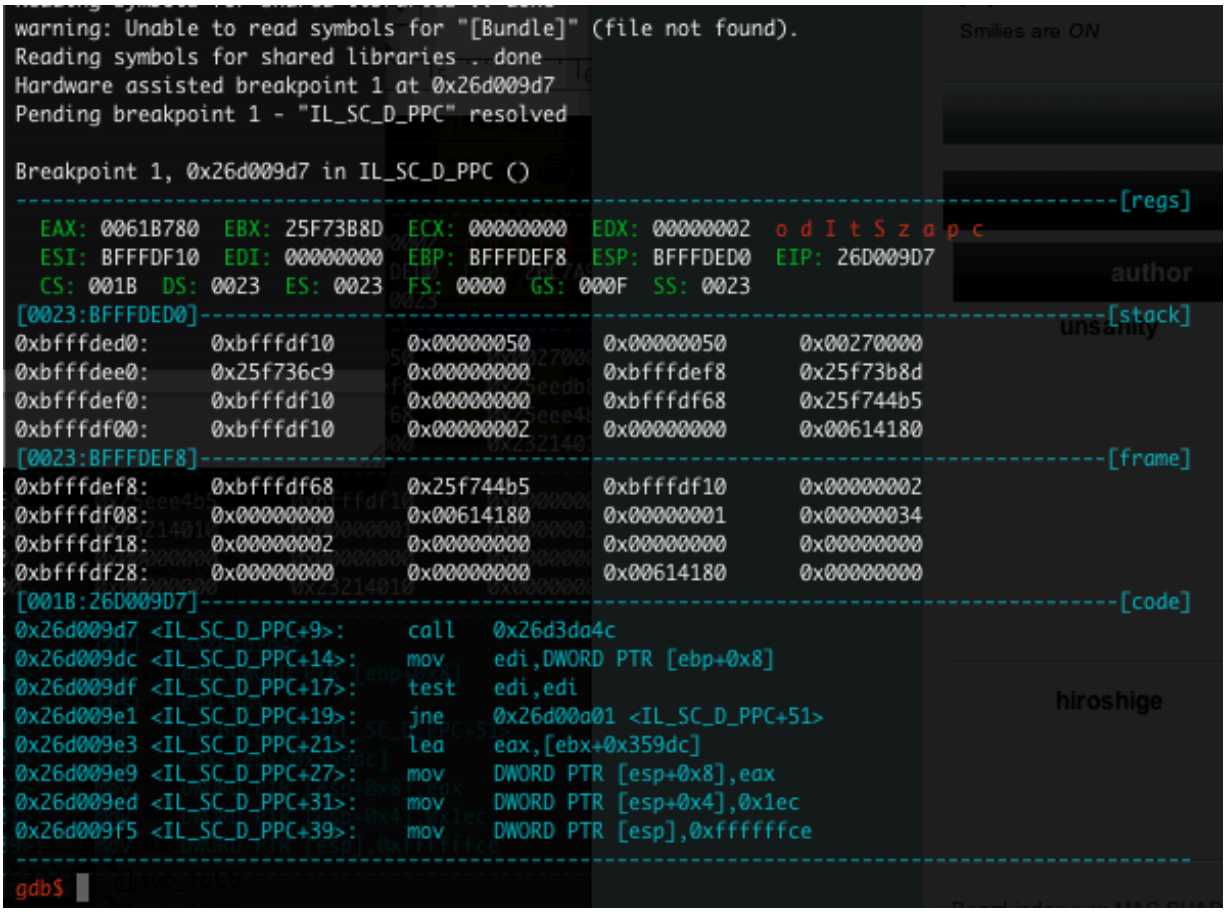
```
gdb$ set breakpoint pending on
gdb$ source /.gdbinit
```

Now we'll go ahead and set a hardware breakpoint and run the host with the "r" command.

Code:

```
gdb$ hb IL_SC_D_PPC
Function "IL_SC_D_PPC" not defined.
Hardware assisted breakpoint 1 (IL_SC_D_PPC) pending.
gdb$ r
```

At this point, AU Lab will probably run normally and GDB will not break. If this happens, just click through the windows in AU Lab (or your other host) and add the plugin as an effect to a track. This will cause the plugin to be instantiated, and GDB should then break on IL_SC_D_PPC.


warning: Unable to read symbols for "[Bundle]" (file not found).
Reading symbols for shared libraries . done
Hardware assisted breakpoint 1 at 0x26d009d7
Pending breakpoint 1 - "IL_SC_D_PPC" resolved
Breakpoint 1, 0x26d009d7 in IL_SC_D_PPC ()
[regs]
EAX: 0061B780 EBX: 25F73B8D ECX: 00000000 EDX: 00000002 o d I t S z a p c
ESI: BFFFDF10 EDI: 00000000 EBP: BFFFDEF8 ESP: BFFFDED0 EIP: 26D009D7
CS: 001B DS: 0023 ES: 0023 FS: 0000 GS: 000F SS: 0023
[0023:BFFFDED0]
[stack]
0xbfffded0: 0xbfffdf10 0x00000050 0x00000050 0x00270000
0xbfffdee0: 0x25f736c9 0x00000000 0xbfffdef8 0x25f73b8d
0xbfffdef0: 0xbfffdf10 0x00000000 0xbfffdf68 0x25f744b5
0xbfffdf00: 0xbfffdf10 0x00000002 0x00000000 0x00614180
[0023:BFFFDEF8]
[frame]
0xbfffdef8: 0xbfffdf68 0x25f744b5 0xbfffdf10 0x00000002
0xbfffdf08: 0x00000000 0x00614180 0x00000001 0x00000034
0xbfffdf18: 0x00000002 0x00000000 0x00000000 0x00000000
0xbfffdf28: 0x00000000 0x00000000 0x00614180 0x00000000
[001B:26D009D7]
[code]
0x26d009d7 <IL_SC_D_PPC+9>: call 0x26d3da4c
0x26d009dc <IL_SC_D_PPC+14>: mov edi,DWORD PTR [ebp+0x8]
0x26d009df <IL_SC_D_PPC+17>: test edi,edi
0x26d009e1 <IL_SC_D_PPC+19>: jne 0x26d00a01 <IL_SC_D_PPC+51>
0x26d009e3 <IL_SC_D_PPC+21>: lea eax,[ebx+0x359dc]
0x26d009e9 <IL_SC_D_PPC+27>: mov DWORD PTR [esp+0x8],eax
0x26d009ed <IL_SC_D_PPC+31>: mov DWORD PTR [esp+0x4],0x1ec
0x26d009f5 <IL_SC_D_PPC+39>: mov DWORD PTR [esp],0xfffffce
gdb$

OK, now we just need to see what led up to this call to PACE. We do that with the backtrace command, or simply "bt":

Code:

```
gdb$ bt
#0  0x26d009d7 in IL_SC_D_PPC ()
#1  0x25f744b5 in JimAUViewEntry ()
#2  0x25f733d1 in JimAUViewEntry ()
#3  0x25f73446 in JimAUViewEntry ()
#4  0x25f73bea in JimAUViewEntry ()
#5  0x25f73c9e in JimAUViewEntry ()
#6  0x25f3d243 in dyld_stub_sprintf ()
#7  0x25f3d553 in dyld_stub_sprintf ()
#8  0x25f6d3c7 in dyld_stub_sprintf ()
#9  0x25fc6b3e in JimAUViewEntry ()
#10 0x25f6d9a4 in JimAUGenericEntry ()
#11 0x93542ce5 in CallComponentDispatch ()
#12 0x93542cc6 in CallComponentOpen ()
#13 0x93540f94 in OpenAComponent ()
#14 0x90b07494 in AudioComponentInstanceNew ()
#15 0x9310ba1b in AudioUnitNodeInfo::Open ()
#16 0x9310e0b4 in AudioUnitGraph::NewNode ()
#17 0x9310e2bd in AUGraphNewNode_Imp ()
#18 0x9310e342 in AUGraphAddNode ()
```

So here we see functions that are common to the loading of all AU plugins, CallComponentOpen, OpenAComponent, CallComponentDispatch, etc. So it looks like the first entry into the plugin is JimAUGenericEntry. I like to verify this is indeed the DUY DaD Tape plugin because there are AU plugins that load a VST plugin and the code you find is actually in the VST plugin, but you've wasted a good 10 minutes searching through the AU plugin before realizing.
We can verify this with the method I gave previously. Leave GDB open and broken on IL_SC_D_PPC. In a separate Terminal window (or tab), type the following command:

Code:

```
nj-69-69-69-196:~ Unsanity$ vmmap -resident -interleaved i386_AU | grep -i
dad
Found process 6753 (i386_AU_Lab) from partial name i386_AU
Found process 6753 (i386_AU_Lab) from partial name i386_AU
```

```
__TEXT                 25f29000-25fef000 [  792K/  640K] r-x/rwx
SM=COW  /Library/Audio/Plug-Ins/Components/DaD Tape
AU.component/Contents/MacOS/DaD Tape AU
__DATA                 25fef000-2647b000 [ 4656K/ 4636K] rw-/rwx
SM=COW  /Library/Audio/Plug-Ins/Components/DaD Tape
AU.component/Contents/MacOS/DaD Tape AU
__DATA                 2647b000-2647e000 [   12K/    8K] rw-/rwx
SM=PRV  /Library/Audio/Plug-Ins/Components/DaD Tape
AU.component/Contents/MacOS/DaD Tape AU
__IMPORT               2647e000-2647f000 [    4K/    4K] rwx/rwx
SM=PRV  /Library/Audio/Plug-Ins/Components/DaD Tape
AU.component/Contents/MacOS/DaD Tape AU
__LINKEDIT             2647f000-2648e000 [   60K/   60K] r--/rwx
SM=COW  /Library/Audio/Plug-Ins/Components/DaD Tape
AU.component/Contents/MacOS/DaD Tape AU
```

We can see that the JimAUGenericEntry address from GDB (0x25f6d9a4) corresponds to something that's in the TEXT segment of our DUY DaD Tape plugin. Keep this backtrace address list handy. If you really wanted to make things easy on yourself, you can calculate the offsets from the backtrace addresses by subtracting the base address from them. For example, you could find the offset of JimAUGenericEntry in the file by taking it's address (0x25f6d9a4) and subtracting the base TEXT segment address from it (0x25f29000) which we found from the vmmap output. If you do this for the 10 address leading up to IL_SC_D_PPC, you'll have a list of file offsets which would make it easier to trace in IDA Pro. Just use Apple's Calculator.app program to do the calculations.

OK, so we've done the original reconnaissance, and we have enough info to finish this out in IDA Pro. Go ahead and copy the binary /Library/Audio/Plug-Ins/Components/DaD Tape AU.component/Contents/MacOS/DaD Tape AU to VMWare if you use it, and load it up in IDA Pro.

gdbinit file I use:
http://www.mediafire.com/file/ij606y31rm80l39/gdbinit

Step 2: Analysis

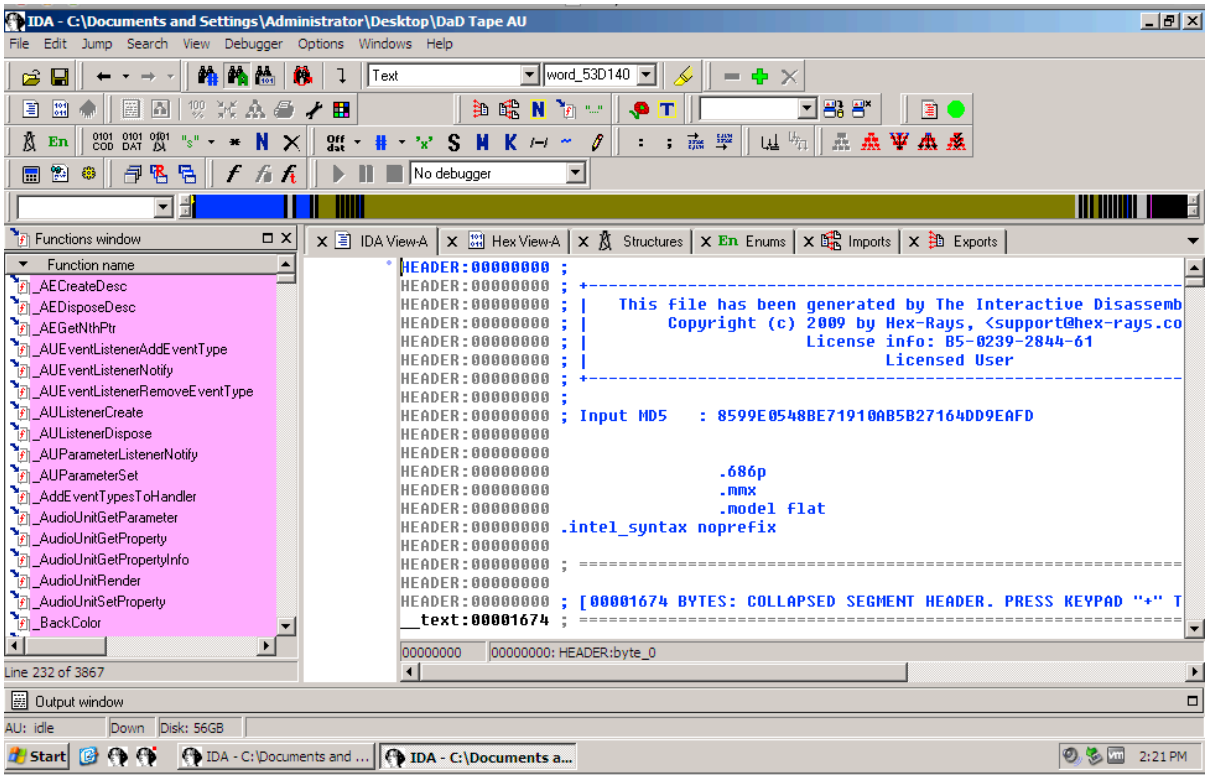
or, ***What the Hell is Going On?***

Additional tools required:

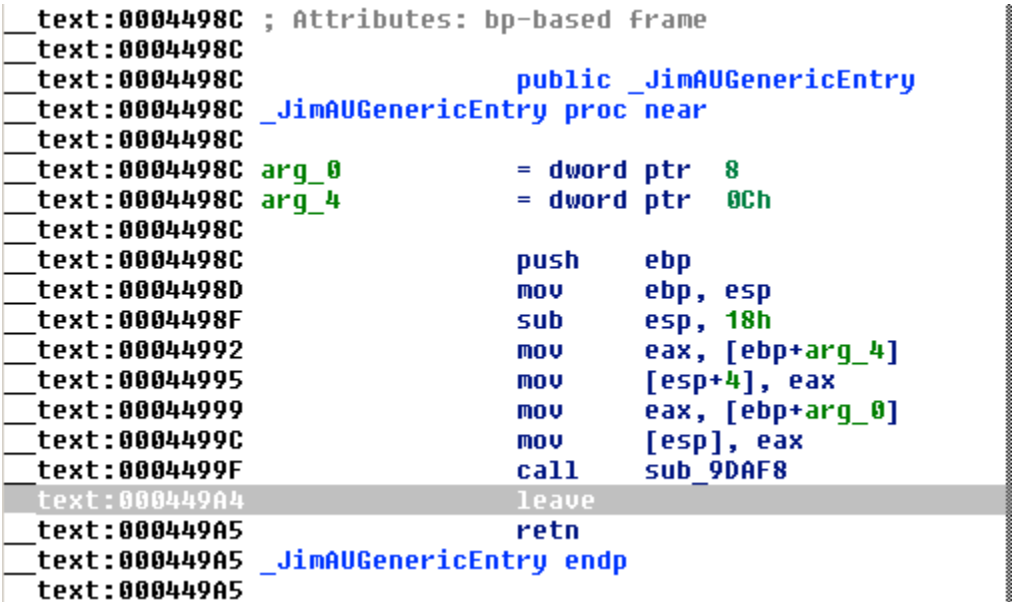
a brain, healthy dose of patience

. Most of this I know just because of experience with several different programs, but I hope that some of you out there can learn these little tricks and apply them elsewhere. The most important tool here is the Hexrays Decompiler. It is an amazing tool that gives you ***readable C pseudocode***, which makes it FAR, FAR easier to understand a program. The power of IDA Pro comes because of the ability to make changes to the output you see, to make it readable. I will attempt to explain and post screenshots of what I've done to make the analysis easier.

OK, so it's loaded up in IDA Pro. On the left, you'll see a list of functions that IDA Pro has found, and the main disassembly window on the right. Click the heading of the function list (the bar up top title "Function name") so that the functions are listed in alphabetical order. To make viewing the important content easier, I drag the bottom bar down and shorten the function list on the right. Here's what my window looks like after I just loaded DaD Tape AU and made these adjustments:



Now look through the function list on the left for the function "JimAUGenericEntry" and double-click on it. The main content view will now show that function. If you look at the offsets on the side, and if you did the calculations to find the offsets from the GDB backtrace, you'll see that 0x449A4 is one we can see here.



Now, it's on the "leave" opcode because the way backtraces work is by showing the instruction that was pushed onto the stack by a "call" opcode, which is the instruction following the call and where execution will return when the call finishes. So all the offsets that we have are actually offsets to the next instruction after a call was made. Knowing this will make it easier to follow the backtrace in IDA Pro.

OK, so before that "leave" command, we see the call that we need to follow, "call sub_9DAF8". Since we can only see the offset in IDA-View (the disassembly view), it's a common practice to toggle back-and-forth between decompile view and this IDA-View. "ESC" is a quick shortcut key to move backwards, as well as the left- and -right arrows that you see up in the toolbar next to the "open" and "save" icons. If you haven't used IDA Pro for much, take a quick second to familiarize yourself with navigating around it. You enter decompilation by pressing "F5". If you've never seen this before, you'll immediately notice that this is far easier than assembly to read. OK, so that "call sub_9DAF8", double-click that to follow it in.

The next offset on your list from the backtrace is 0x9DB3E. Now we see that one here follows a "call sub_442C6". Go ahead here and press "F5" so we can see the C code for this function before we follow this call. Here is what you should see:

```
int __cdecl sub_9DAF8(int a1, int a2)
{
  int v2; // ST2C_4@2
  void *v3; // ST1C_4@2
  int v5; // [sp+28h] [bp-20h]@1

  v5 = 0;
  if ( *(_WORD *)(a1 + 2) == -1 )
  {
    v2 = *(_DWORD *)(a1 + 4);
    v3 = (void *)operator new(276);
    sub_442C6(v3, v2);
    (*(void (__cdecl **)(void *))(*(_DWORD *)v3 + 12))(v3);
    SetComponentInstanceStorage(v2, v3);
  }
  else
  {
    v5 = sub_40264(a1, a2);
  }
  return v5;
}
```

Now I'll try to explain what I can about the nuances of c++ code. Here we see a call to operator new(276), which gets returned in v3, which in turn is passed to sub_442C6 as the first argument. Now what this typically means is that a new c++ class is being initialized. For example, any of you familiar with writing c++ code, this is what it's kind of like. Say you have a class "Foo" defined, and a variable called "bar" and a function called "func". There are two ways to instantiate a class in c++, on the stack, or on the heap. The following illustrates a stack initialization, and how the different members are accessed.

Code:

```
Foo fooVariable;
fooVariable.bar = 12345;
fooVariable.func();
```

And here is it initialized on the heap with a pointer returned:

Code:

```
Foo fooVariable = new Foo();
fooVariable->bar = 12345;
fooVariable->func();
```

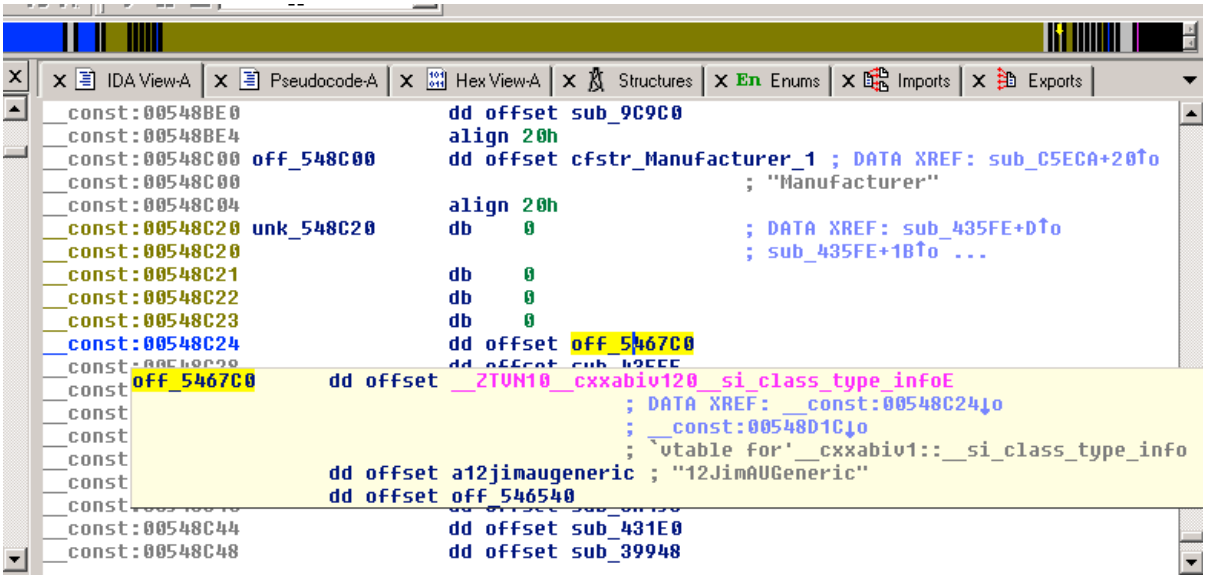
When you see an operator new in IDA Pro and the pointer that returned get passed into a function, it's

probably a class initialization like we see in the second example. So let's go ahead and follow sub_44C26 and see what's done with the new pointer passed in.

```
char __cdecl sub_442C6(void *a1, int a2)
{
  int v2; // edx@2
  int v3; // ecx@2
  char result; // al@6
  __int16 v5; // [sp+2Ch] [bp-Ch]@3
  char v6; // [sp+2Eh] [bp-Ah]@3

  sub_3A096(a1, a2, 1);
  *(_DWORD *)a1 = (char *)&unk_548C20 + 8;
  *((_DWORD *)a1 + 2) = (char *)&unk_548C20 + 256;
  sub_95E10((char *)a1 + 268);
  *((_DWORD *)a1 + 63) = -1;
  *((_DWORD *)a1 + 66) = 0;
  *((_DWORD *)a1 + 65) = 0;
  *((_DWORD *)a1 + 64) = 0;
  if ( !dword_53D144 )
  {
    v2 = CFBundleGetBundleWithIdentifier(&cfstr_Com_duy_au_plu);
    dword_53D144 = v2;
    if ( dword_53D144 )
```

OK, here we can definitely see that it's a class being instantiated, so we know now that we are actually in the class "constructor" (in our previous example, that would be Foo()). Here is how we can tell that. You'll notice that before, the operator new returned a pointer to v3, and v3 was passed into sub_3A096, so v3 is now the first argument, or the "void *a1" you see up top. Now you'll see that the second instruction assigns the address of unk_548C20 to what a1 points to. Lets double-click unk_548C20 and see what that is. Once you get there, you see a list of a bunch of functions. This is the classes "v-table", or virtual table, basically a list of all functions that belong to the class being instantiated. the first one listed is a pointer to the class name, among other things. So here is what information we can gather from that:



You can see from the screenshot that if I hold my mouse over that first offset in the v-table list, it pops up a little window showing me what's at that address, and you can see "JimAUGeneric" there. This is the class name that belongs to the v-table here, and this v-table is getting assigned to our first argument, so now we now that that is in fact a class called JimAUGeneric being instantiated.
Don't worry, these concepts took me a while to understand on my own, and you will only FULLY understand them with experience.

OK, so continue following the offsets we saw in the backtrace and checking the function out by pressing "F5" before continuing on to the next one. When you get to the offset 0x14243 and enter the call before it, check out the decompilation by pressing "F5". Whoa! Here we see some clear text "/Users/macbuilder/Builds/cvs_bleed/pace_mac/PACE/PACEInterface/PACEInterface.c". So we know now that this function is actually the beginning of the call into PACE.

```
signed int __cdecl sub_4AC36(__int16 a1, int a2)
{
  signed int result; // eax@2
  int v3; // esi@3
  int v4; // ecx@3

  if ( a2 )
  {
    *(_DWORD *)a2 = 0;
    sub_4B1AC(a1 == -1, -50, 4212, "/Users/macbuilder
    v3 = sub_4A12E(a1);
    sub_4AB7C(v4, 0);
    *(_DWORD *)a2 = v3;
    result = 0;
  }
  else
  {
    result = -50;
  }
  return result;
}
```

OK, from looking at this function now and poking around a little bit (and if we have any experience here at all) we can see that there are a couple of possible values to be returned. -50 is one, and 0 is the other. We can deduce that -50 means an error occured, and 0 means that everything was successful. We see the function sub_4A12E here, and if we look inside it, we see a call to NewPtrClear, which is basically a malloc call, that sets v1. Some stuff is done with v1, and then v1 is returned.

```
int __cdecl sub_4A12E(__int16 a1)
{
  int v1; // edi@1
  __int16 v2; // ax@2
  char v4; // [sp+30h] [bp-B8h]@2
  char v5; // [sp+80h] [bp-68h]@2

  v1 = NewPtrClear(260);
  sub_4B22A(v1, 3574, "/Users/macbuilder/Builds/cvs_bleed/pace_mac/PACE/PACEInterfa
  *(_WORD *)(v1 + 176) = -1;
  if ( a1 != -1 )
  {
    v2 = FSGetForkCBInfo(a1, 0, 0, 0, 0, &v5, 0);
    sub_4B06A(v2);
    memcpy(&v4, &v5, 0x50u);
    memcpy((void *)(v1 + 12), &v5, 0x50u);
    memcpy((void *)(v1 + 92), &v4, 0x50u);
  }
  *(_DWORD *)v1 = v1;
  *(_DWORD *)(v1 + 4) = 0;
  *(_DWORD *)(v1 + 8) = 260;
  *(_DWORD *)(v1 + 172) = 0;
  *(_WORD *)(v1 + 176) = a1;
  *(_BYTE *)(v1 + 178) = 0;
  return v1;
}
```

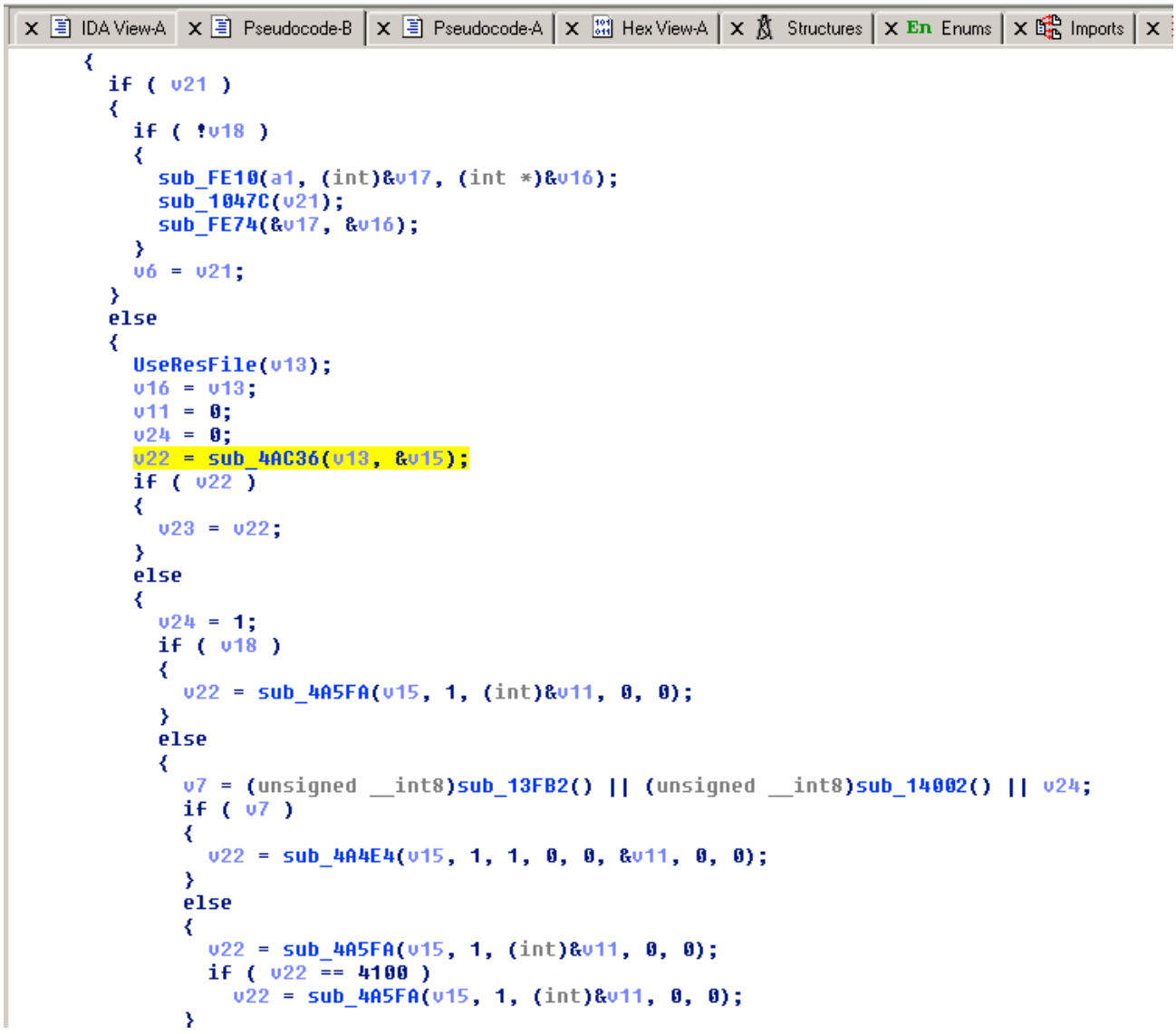
v1 is returned to v3 in sub_4AC36, and v3 is assigned to a2, or the second argument.

```
signed int __cdecl sub_4AC36(__int16 a1, int a2)
{
  signed int result; // eax@2
  int v3; // esi@3
  int v4; // ecx@3

  if ( a2 )
  {
    *(_DWORD *)a2 = 0;
    sub_4B1AC(a1 == -1, -50, 4212, "/Users/macbuilder
    v3 = sub_4A12E(a1);
    sub_4AB7C(v4, 0);
    *(_DWORD *)a2 = v3;
    result = 0;
  }
  else
  {
    result = -50;
  }
  return result;
}
```

So we can see that there's a call made to PACE, and PACE allocates a pointer that holds some information, which gets assigned to a variable that's passed in. OK. Be sure you at least kind of understand this part. So on success, 0 is returned and the second argument holds a pointer to some PACE structure. So now let's see what argument 2 was passed in.

This call was actually made from sub_14058, and is right before the backtrace offset 0x1428F. So find this offset and then press "F5" to enter decompilation. You should be brought to this location, the call that leads to where we were at is highlighted in yellow:



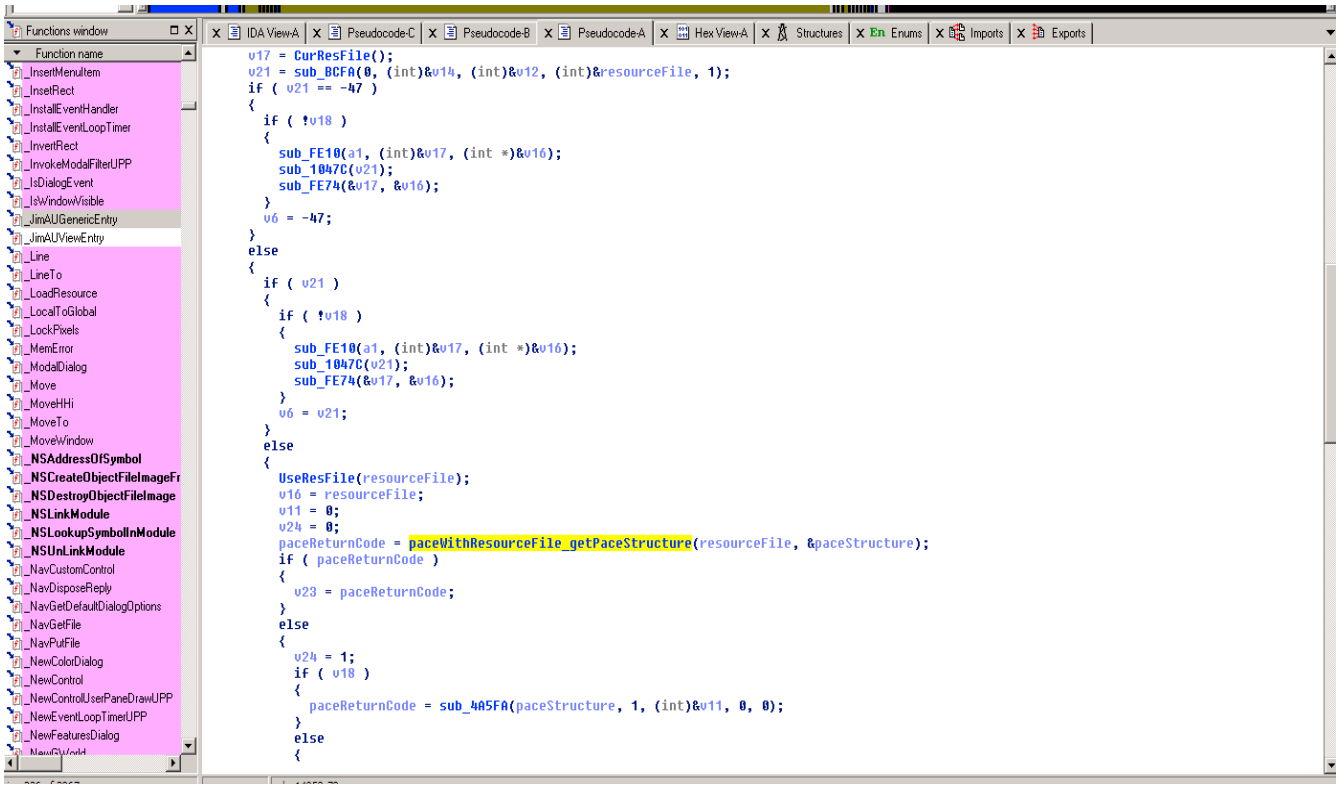
So the argument 2 that is assigned a pointer to the PACE structure is v15, and the result code is assigned to v22. Make sure that you understand where we are at now before I go on. If you have questions, you can ask them here, but please stay specific to this step and what we've done so far.

Step 3: Further Analysis

or, ***What the Hell Does That Mean?***

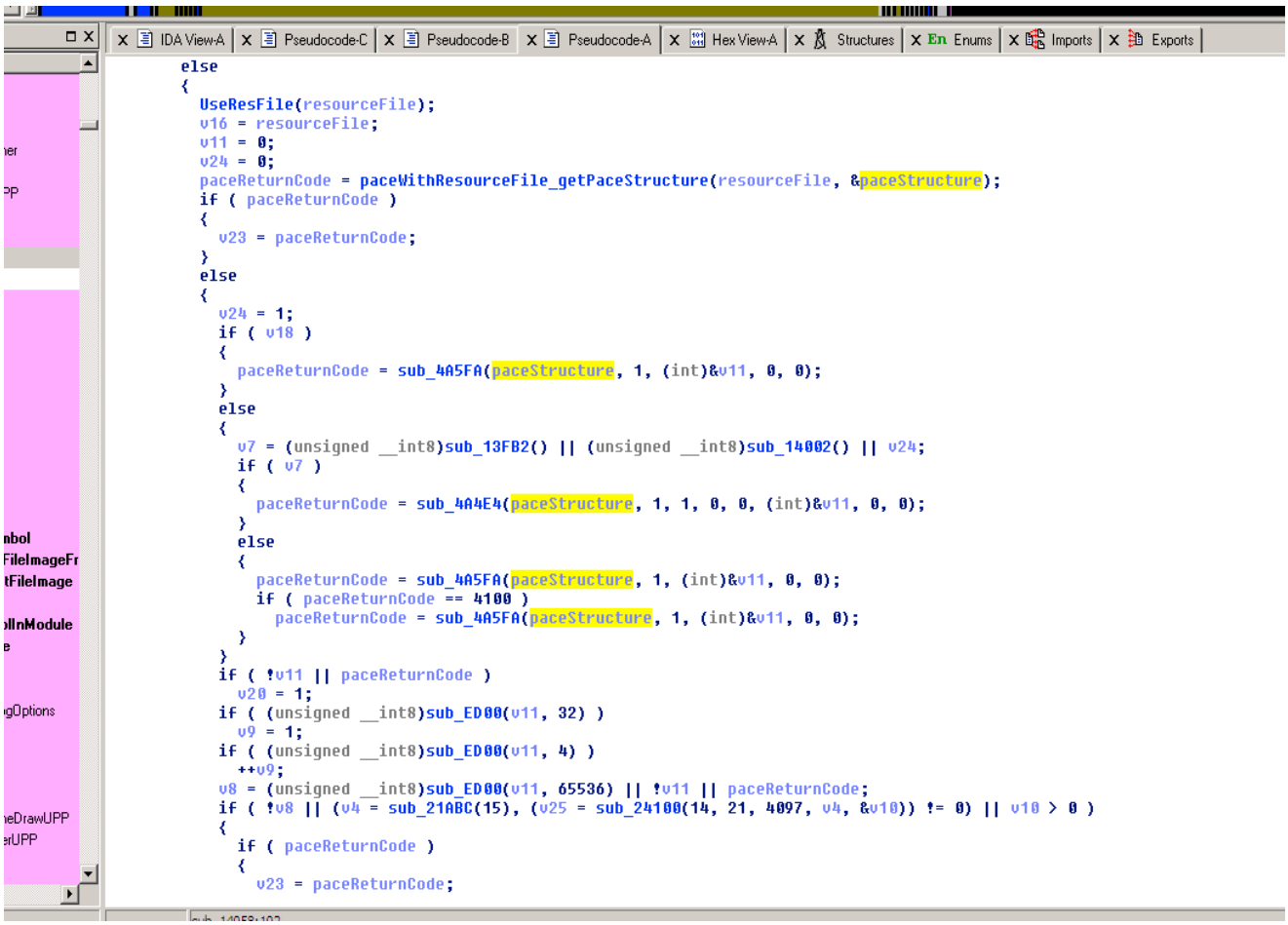
Additional tools required:
a sharp brain, some coffee

OK, to pick up where we left off yesterday morning. We had identified where there seemed to be a call to PACE, that returns a new structure in the second argument passed in, and also a return code to indicate success or failure. You can begin to make minor adjustments to the IDA output to make it easier to remember what happened by right-clicking the name and choosing "rename item", or pressing "N". IDA is ultimately customizable in how you can manipulate the output. So with that minor change made, here is that same function:

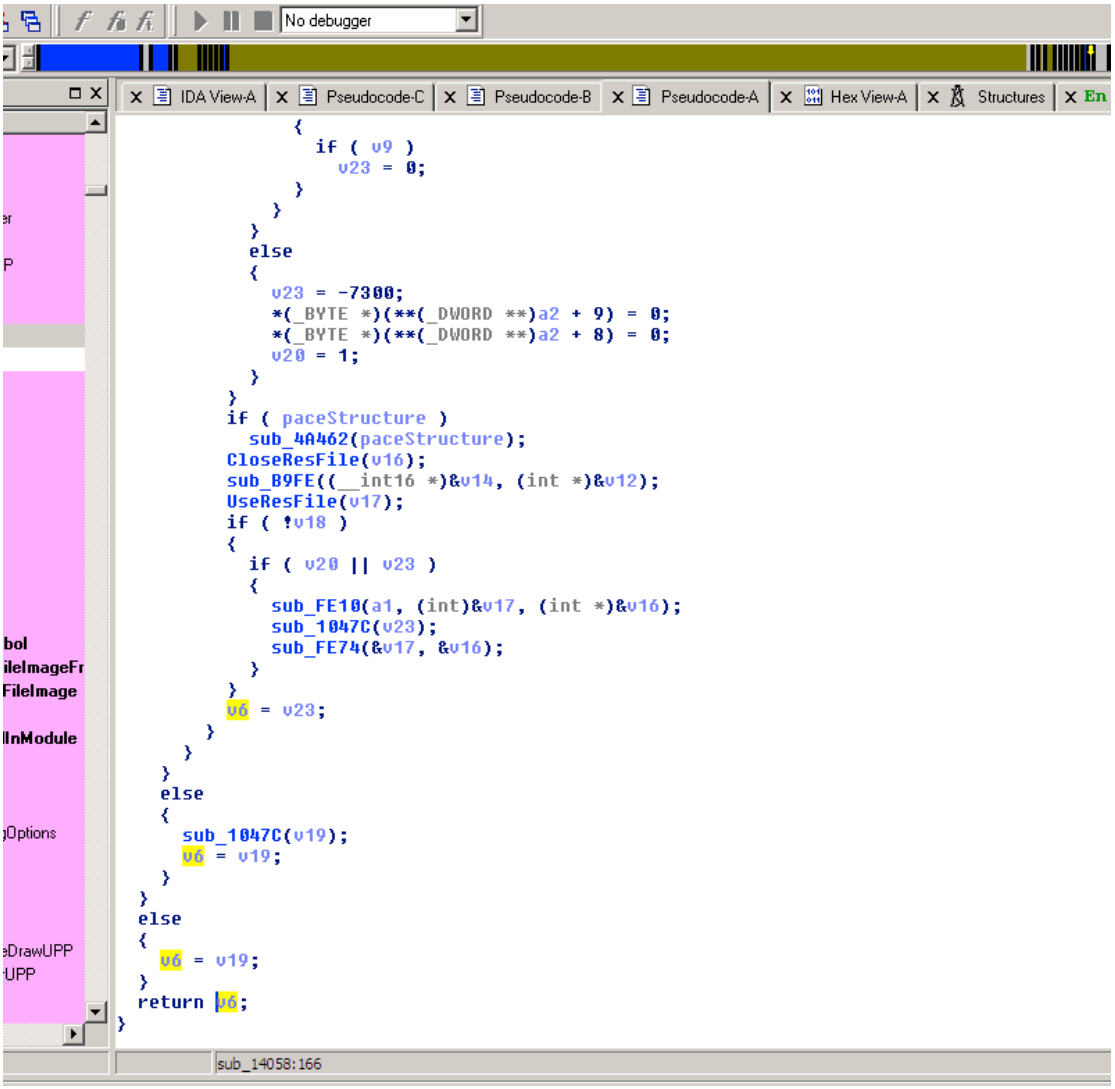


Notice how I filled in the resourceFile argument too. I saw the UseResFile above, and then a function you can see up top in the screenshot that actually passes back the resource file. I don't know much else about it, but as soon as you figure something out, it's a good idea to fill it in in IDA so you can read it much easier. So now we have to see what is done with the PACE structure that gets returned, and if it's used other place. Most of the time you won't just be able to NOP it out, because that structure is expected to be there somewhere else in the program, so you'll have to follow it and see. So for now, just click on the

paceStructure variable we have filled in, and all occurrences will be highlighted in yellow, and we can see what's done with it.



After looking, you can see quite clearly that this structure never leaves this function. Hmmm, matter of fact, it appears that it's existence only affects what value gets assigned to the return code that is eventually returned.

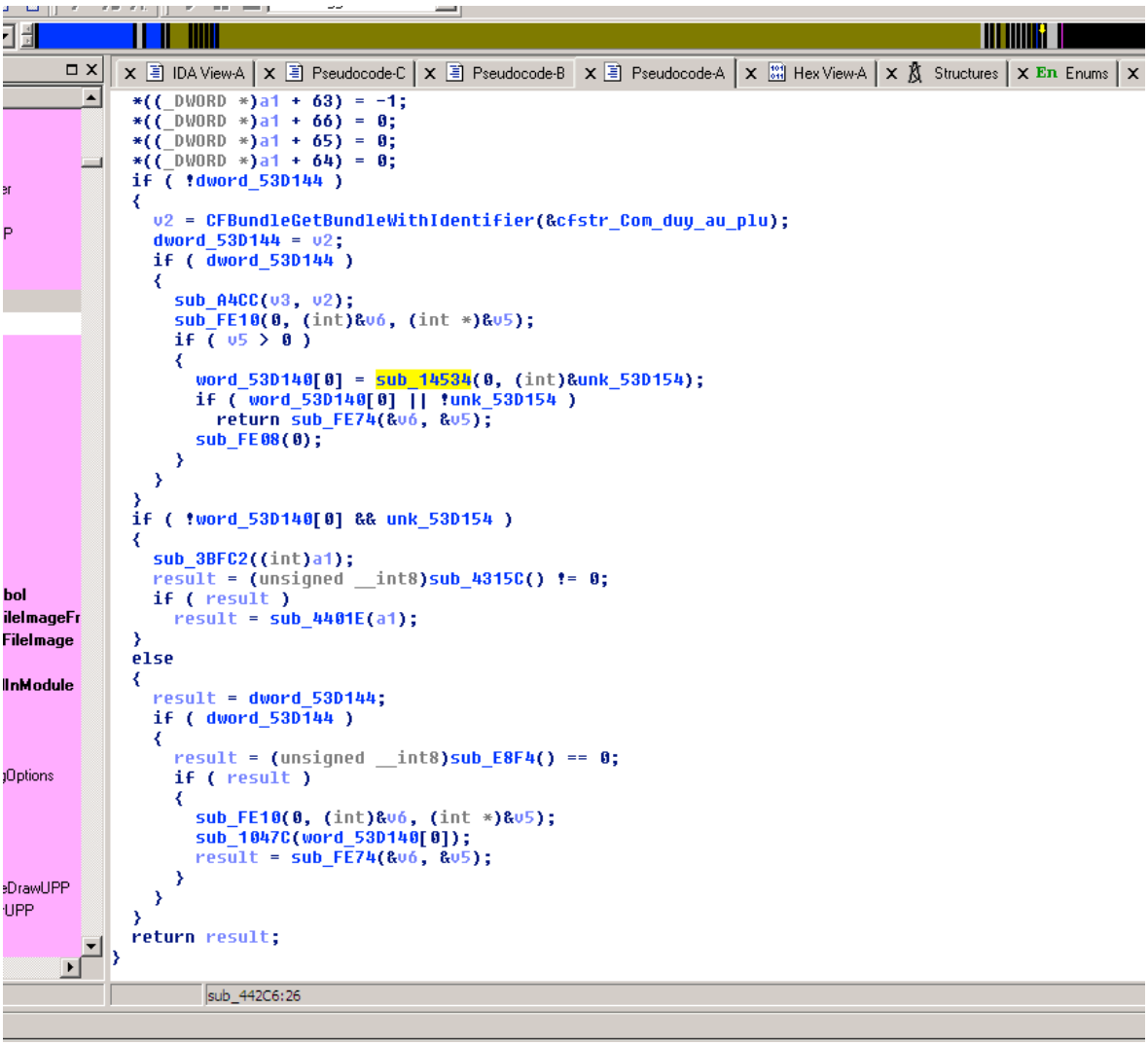



```
                {
                  if ( v9 )
                    v23 = 0;
                }
              }
            }
            else
            {
              v23 = -7300;
              *(_BYTE *)(**(_DWORD **)a2 + 9) = 0;
              *(_BYTE *)(**(_DWORD **)a2 + 8) = 0;
              v20 = 1;
            }
          }
          if ( paceStructure )
            sub_4A462(paceStructure);
          CloseResFile(v16);
          sub_B9FE((__int16 *)&v14, (int *)&v12);
          UseResFile(v17);
          if ( !v18 )
          {
            if ( v20 || v23 )
            {
              sub_FE10(a1, (int)&v17, (int *)&v16);
              sub_1047C(v23);
              sub_FE74(&v17, &v16);
            }
          }
          v6 = v23;
        }
      }
    }
    else
    {
      sub_1047C(v19);
      v6 = v19;
    }
  }
  else
  {
    v6 = v19;
  }
  return v6;
}
```

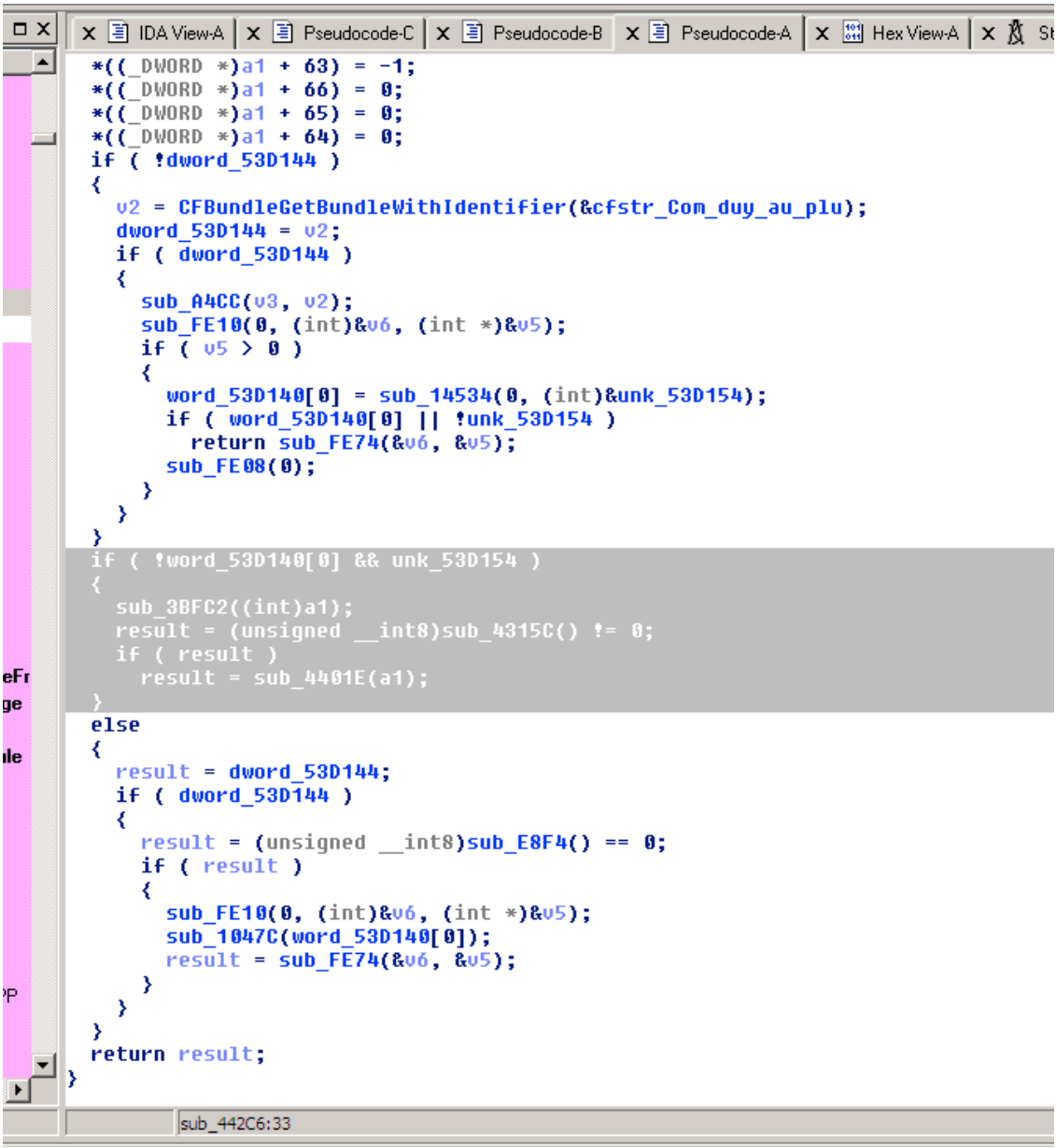
Here v6 is returned, but you can see some other variables get assigned to v6, and the value of those variables depends on some conditional branches in this function. Hmm, so let's see what that return code affects... Press 'ESC' to back up until it takes you to the prior function. If it doesn't take you to the prior function, start again at the JimAUGenericEntry function (in the function list on the left) and follow calls until you get here, and then press 'ESC' to go back to the one prior.

```
__int16 __cdecl sub_14534(int a1, int a2)
{
  return sub_14058(a1, a2);
}
```

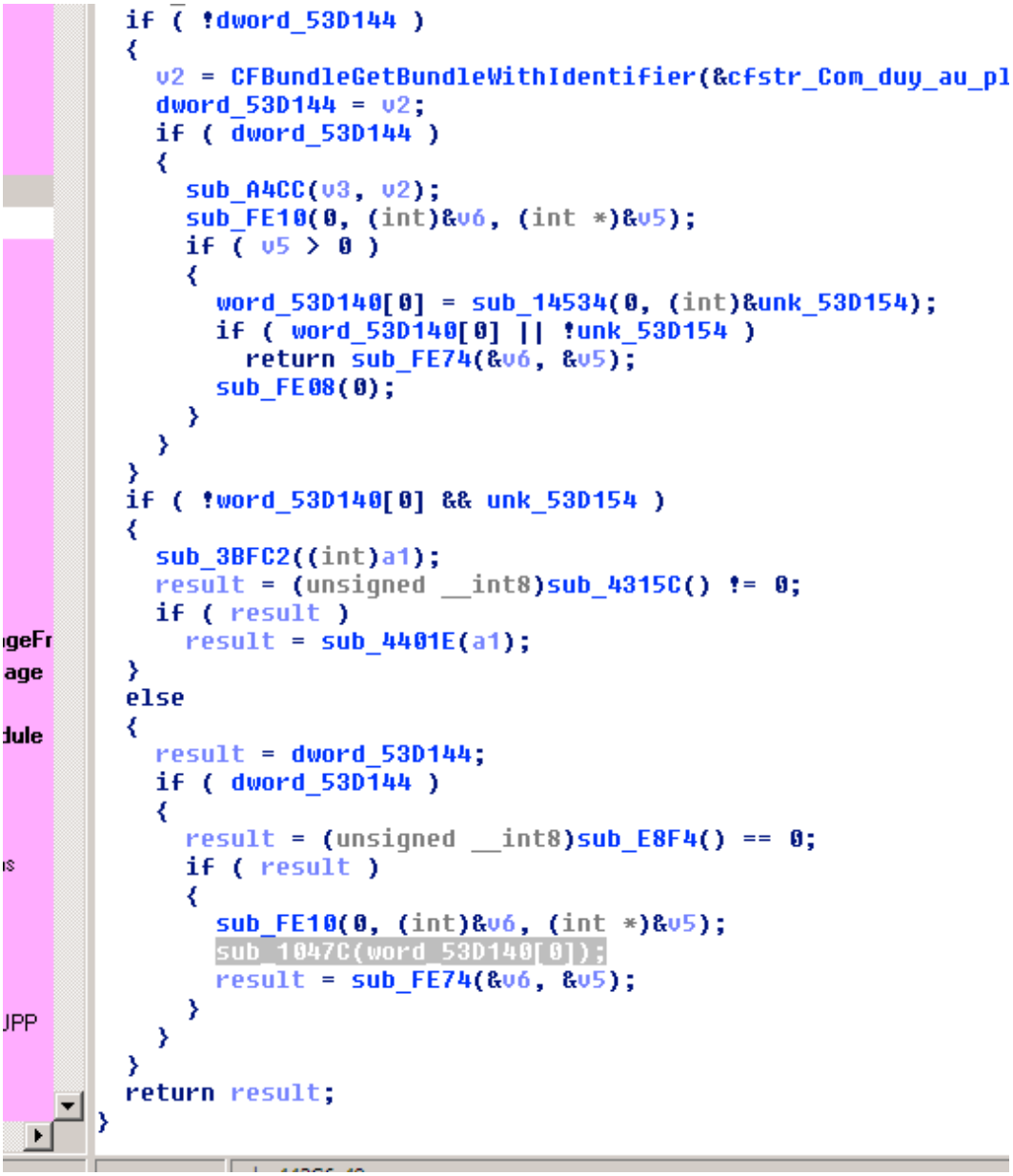
Ok Looks like we need to back up one more...



Alright, so we're back to this function, and we see that the result get's assigned to word_53D140[0], just basically means it's getting assigned to a global variable in the program. Also, if you look, the second variable that gets passed in, unk_53D154, has a "&" in front of it, which means that the *address* of that variable is being passed, or that it's being passed by *reference*. This is usually another way of getting information out of a function. So we'll remember to look at that later when we're back in the other function. OK, but let's look at what's going on in this function.

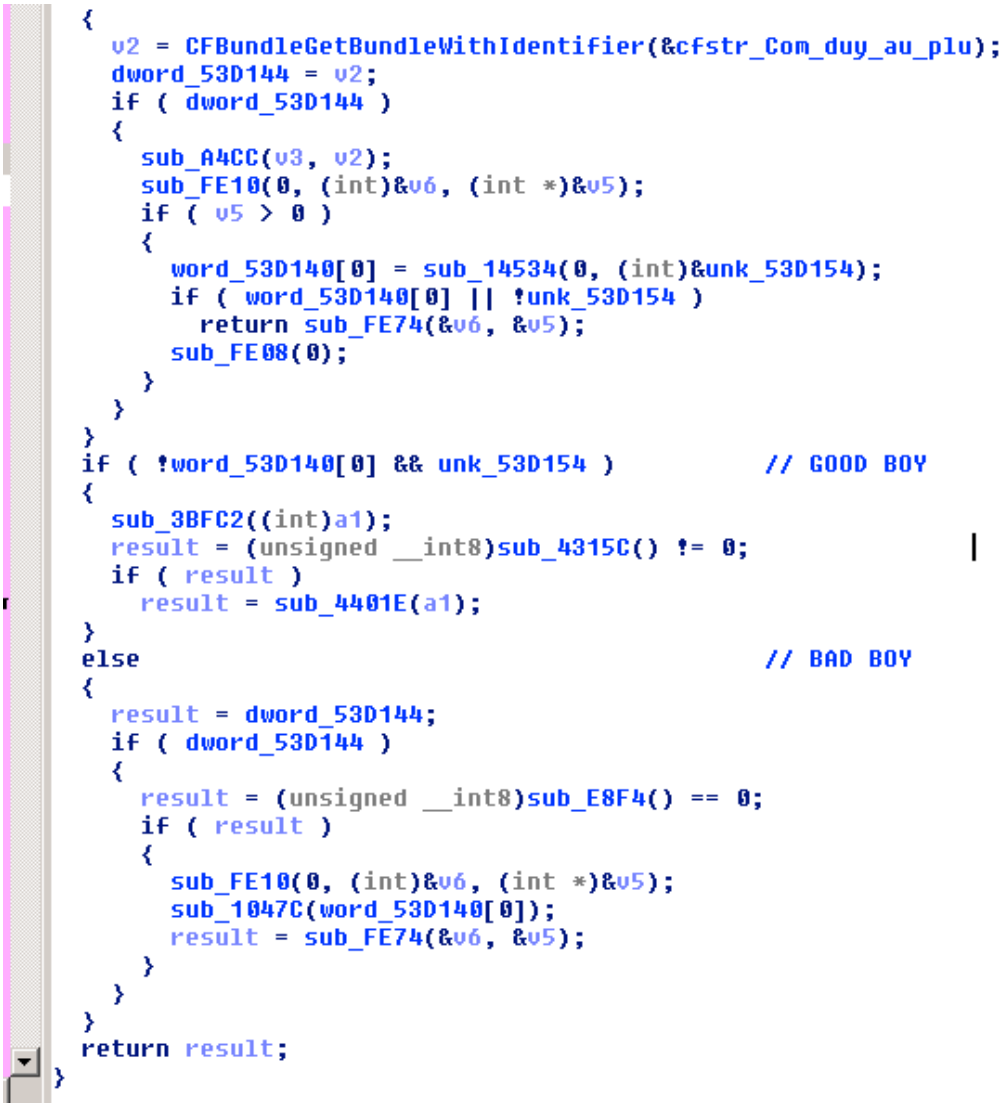


You'll notice what I highlighted in gray; it checks both of those returns from sub_14534, word_53D140[0] and unk_53D154. So it basically reads like this:
If the paceReturnCode is zero and unkVariable is not zero, then process blah blah
Hmm, interesting. Also, if you poke around a bit more, you'll see that word_53D140[0] gets passed into a function near the botton, sub_1047C.



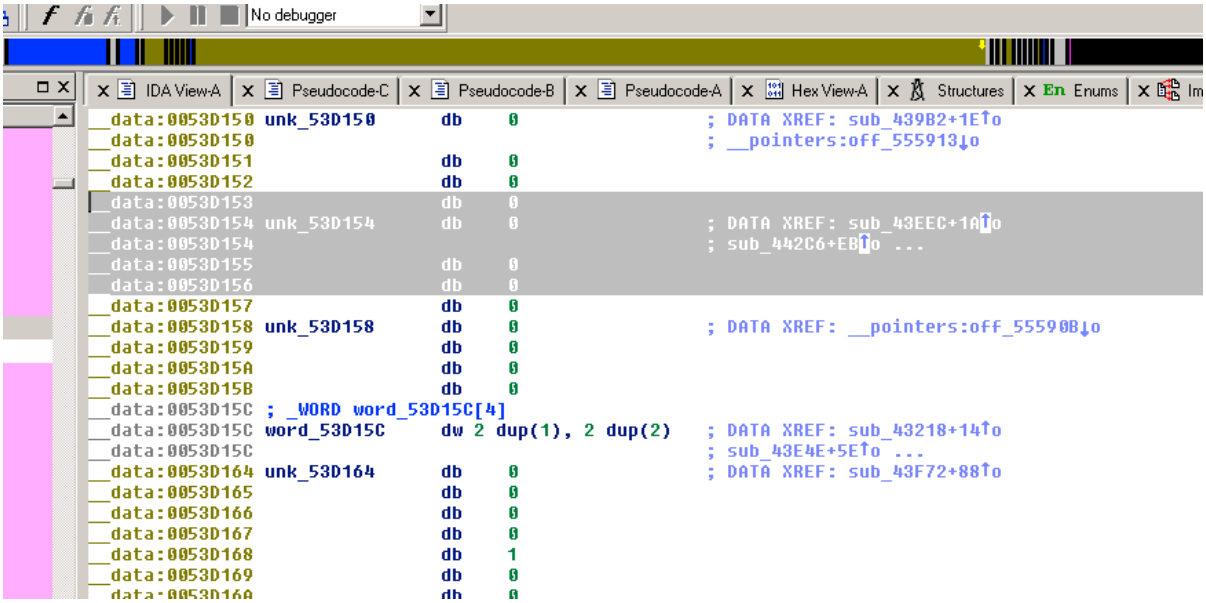
```
if ( !dword_53D144 )
{
  v2 = CFBundleGetBundleWithIdentifier(&cfstr_Com_duy_au_pl
  dword_53D144 = v2;
  if ( dword_53D144 )
  {
    sub_A4CC(v3, v2);
    sub_FE10(0, (int)&v6, (int *)&v5);
    if ( v5 > 0 )
    {
      word_53D140[0] = sub_14534(0, (int)&unk_53D154);
      if ( word_53D140[0] || !unk_53D154 )
        return sub_FE74(&v6, &v5);
      sub_FE08(0);
    }
  }
}
if ( !word_53D140[0] && unk_53D154 )
{
  sub_3BFC2((int)a1);
  result = (unsigned __int8)sub_4315C() != 0;
  if ( result )
    result = sub_4401E(a1);
}
else
{
  result = dword_53D144;
  if ( dword_53D144 )
  {
    result = (unsigned __int8)sub_E8F4() == 0;
    if ( result )
    {
      sub_FE10(0, (int)&v6, (int *)&v5);
      sub_1047C(word_53D140[0]);
      result = sub_FE74(&v6, &v5);
    }
  }
}
return result;
}
```

If you follow that function in, you'll see stuff like "GetDlgItemText" and "GetNewDialog". Hmmm, perhaps that's the error we got earlier when we first ran AU Lab and the dialog popped up saying it's not authorized... So for a preliminary analysis (we'll still need to verify it) we have something like this (to use some of the windows cracking jargon of "good boy" and "bad boy")

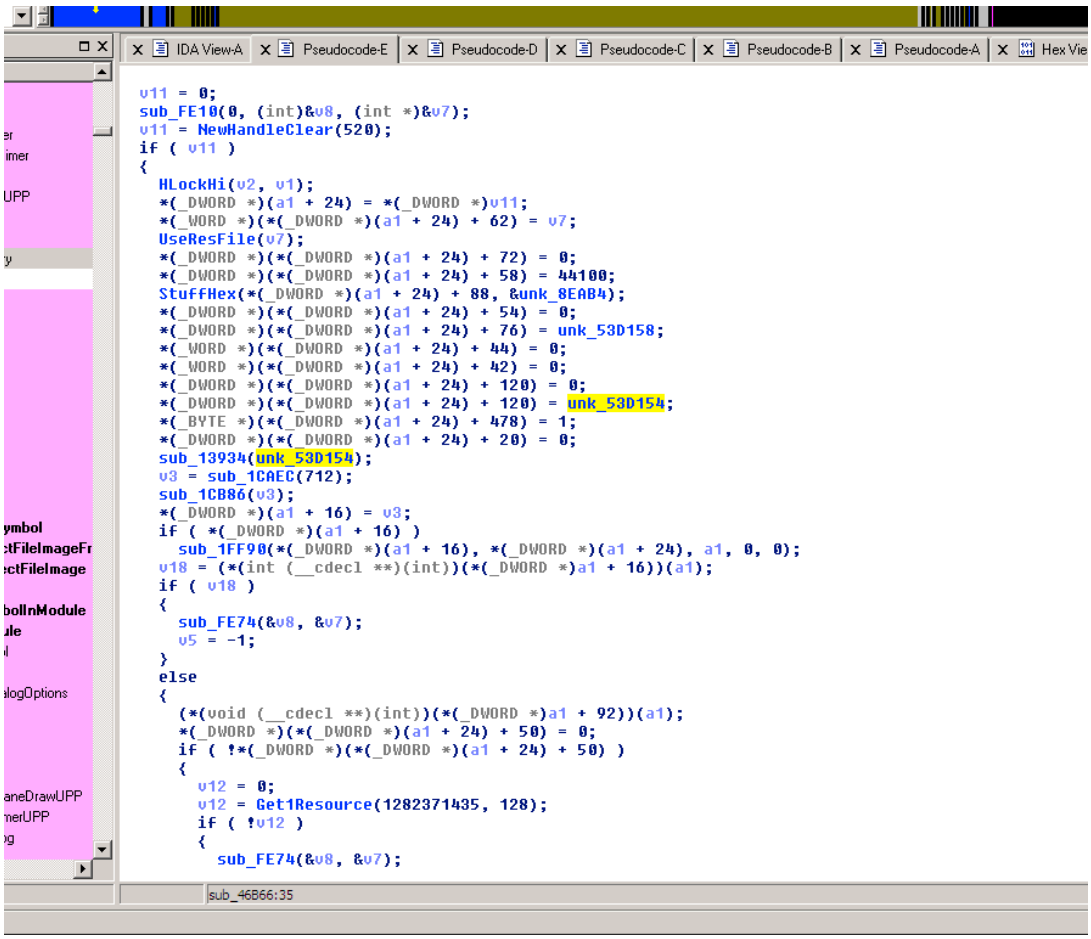


```
{
  v2 = CFBundleGetBundleWithIdentifier(&cfstr_Com_duy_au_plu);
  dword_53D144 = v2;
  if ( dword_53D144 )
  {
    sub_A4CC(v3, v2);
    sub_FE10(0, (int)&v6, (int *)&v5);
    if ( v5 > 0 )
    {
      word_53D140[0] = sub_14534(0, (int)&unk_53D154);
      if ( word_53D140[0] || !unk_53D154 )
        return sub_FE74(&v6, &v5);
      sub_FE08(0);
    }
  }
}
if ( !word_53D140[0] && unk_53D154 )          // GOOD BOY
{
  sub_3BFC2((int)a1);
  result = (unsigned __int8)sub_4315C() != 0;
  if ( result )
    result = sub_4401E(a1);
}
else                                          // BAD BOY
{
  result = dword_53D144;
  if ( dword_53D144 )
  {
    result = (unsigned __int8)sub_E8F4() == 0;
    if ( result )
    {
      sub_FE10(0, (int)&v6, (int *)&v5);
      sub_1047C(word_53D140[0]);
      result = sub_FE74(&v6, &v5);
    }
  }
}
return result;
}
```

You can add comments to the output by clicking on the line you want to add a comment, and pressing the forward-slash (/). OK, so if the first branch is the good boy, then that means that in order to get there, word_53D140 must be zero, and unk_53D154 must contain a value that is not zero (NULL). Hmm, here we can check where else these variables are used in the program... go ahead and double-click on unk_53D154. It'll take you here:



Now, on the right side, you'll see DATA XREF: and some functions... these are cross-references. IDA has identified all the other places in the code where this particular variable is referenced. Pretty handy, huh?! You can follow these back to their source by right clicking in the DATA XREF area, and choosing "Jump to cross-reference...". There will be a few of them, so double-click one in the list, and when it takes you to the place, press "F5" to enter decompilation. Once you've looked at it, you can just back out by pressing 'ESC', or if you want to keep that particular window open, by clicking on one of the other tabs up top (IDA-View to go back to the variable to find other DATA XREFS). A trick here is if you look at all the cross-references for this variable, you'll notice at the bottom of the list is one that says __pointers:off_55590F... If you follow that one back, you'll see another variable that points to unk_53D154 called off_55590F... Be sure to follow these cross-references too. When you do follow those, you'll find some a function that looks like this:


IDA View-A
Pseudocode-E
Pseudocode-D
Pseudocode-C
Pseudocode-B
Pseudocode-A
Hex Vie
v11 = 0;
sub_FE10(0, (int)&v8, (int *)&v7);
v11 = NewHandleClear(520);
if (v11)
{
HLockHi(v2, v1);
*(_DWORD *)(a1 + 24) = *(_DWORD *)v11;
*(_WORD *)(*(_DWORD *)(a1 + 24) + 62) = v7;
UseResFile(v7);
*(_DWORD *)(*(_DWORD *)(a1 + 24) + 72) = 0;
*(_DWORD *)(*(_DWORD *)(a1 + 24) + 58) = 44100;
StuffHex(*(_DWORD *)(a1 + 24) + 88, &unk_8EAB4);
*(_DWORD *)(*(_DWORD *)(a1 + 24) + 54) = 0;
*(_DWORD *)(*(_DWORD *)(a1 + 24) + 76) = unk_53D158;
*(_WORD *)(*(_DWORD *)(a1 + 24) + 44) = 0;
*(_WORD *)(*(_DWORD *)(a1 + 24) + 42) = 0;
*(_DWORD *)(*(_DWORD *)(a1 + 24) + 120) = 0;
*(_DWORD *)(*(_DWORD *)(a1 + 24) + 120) = unk_53D154;
*(_BYTE *)(*(_DWORD *)(a1 + 24) + 478) = 1;
*(_DWORD *)(*(_DWORD *)(a1 + 24) + 20) = 0;
sub_13934(unk_53D154);
v3 = sub_1CAEC(712);
sub_1CB86(v3);
*(_DWORD *)(a1 + 16) = v3;
if (*(_DWORD *)(a1 + 16))
sub_1FF90(*(_DWORD *)(a1 + 16), *(_DWORD *)(a1 + 24), a1, 0, 0);
v18 = (*(int (__cdecl **)(int))(*(_DWORD *)a1 + 16))(a1);
if (v18)
{
sub_FE74(&v8, &v7);
v5 = -1;
}
else
{
(*(void (__cdecl **)(int))(*(_DWORD *)a1 + 92))(a1);
*(_DWORD *)(*(_DWORD *)(a1 + 24) + 50) = 0;
if (!*(_DWORD *)(*(_DWORD *)(a1 + 24) + 50))
{
v12 = 0;
v12 = Get1Resource(1282371435, 128);
if (!v12)
{
sub_FE74(&v8, &v7);
sub_46B66:35

Ahh, if you follow sub_13943 here, you'll see that this variable is accessed, so it needs to be a valid memory location. Bummer, we can't just simply assign it a value. OK armed with this knowledge, go ahead and back up to the original function here where these values are checked.

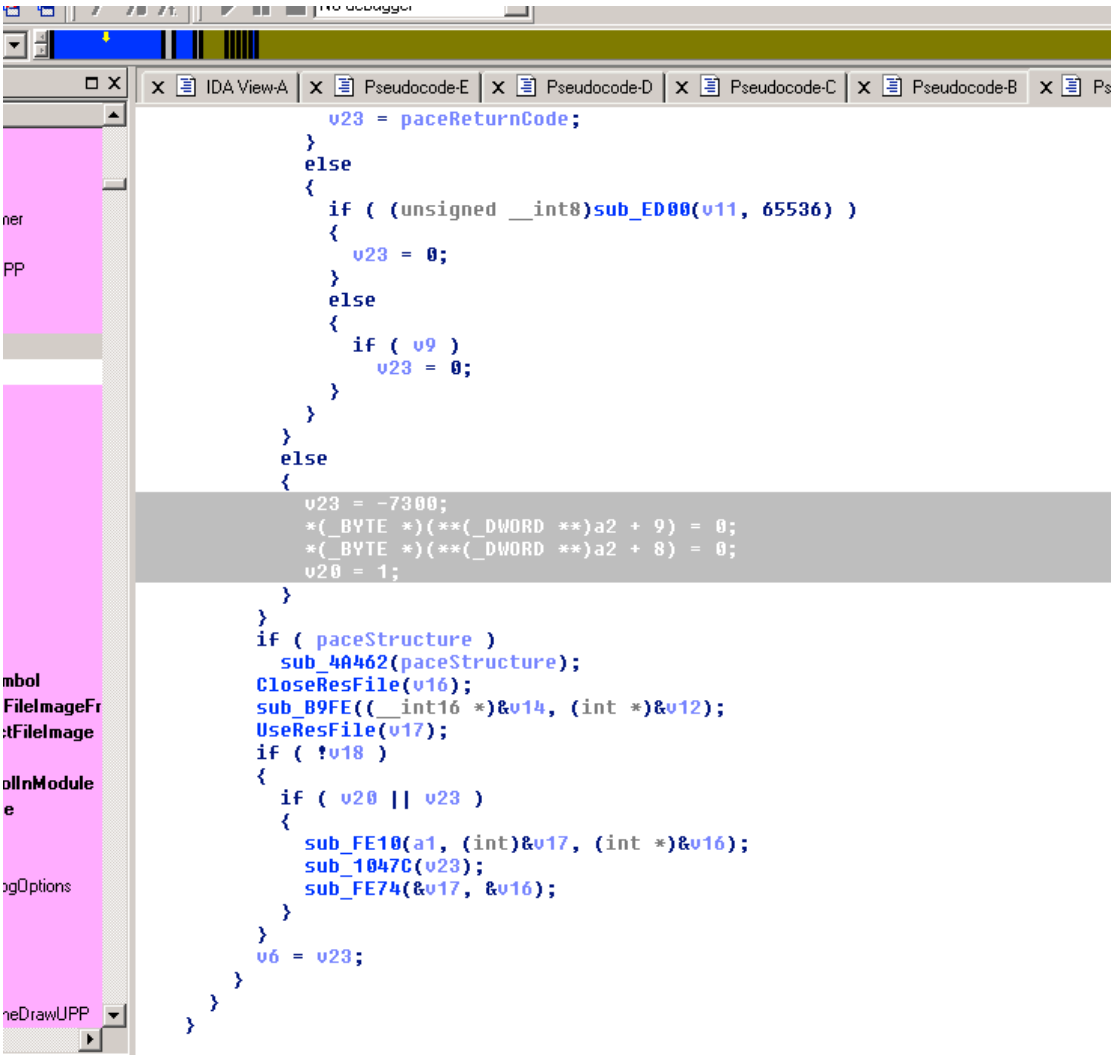
```
{
  v2 = CFBundleGetBundleWithIdentifier(&cfstr_Com_duy_au_plu);
  dword_53D144 = v2;
  if ( dword_53D144 )
  {
    sub_A4CC(v3, v2);
    sub_FE10(0, (int)&v6, (int *)&v5);
    if ( v5 > 0 )
    {
      word_53D140[0] = sub_14534(0, (int)&unk_53D154);
      if ( word_53D140[0] || !unk_53D154 )
        return sub_FE74(&v6, &v5);
      sub_FE08(0);
    }
  }
}
if ( !word_53D140[0] && unk_53D154 )          // GOOD BOY
{
  sub_3BFC2((int)a1);
  result = (unsigned __int8)sub_4315C() != 0;
  if ( result )
    result = sub_4401E(a1);
}
else                                          // BAD BOY
{
  result = dword_53D144;
  if ( dword_53D144 )
  {
    result = (unsigned __int8)sub_E8F4() == 0;
    if ( result )
    {
      sub_FE10(0, (int)&v6, (int *)&v5);
      sub_1047C(word_53D140[0]);
      result = sub_FE74(&v6, &v5);
    }
  }
}
return result;
}
```

Now, step into the function (sub_14534), keeping in mind that the returnCode needs to be zero, and that the second argument passed in will need to be assigned a value.

OK, back in that function, you should begin to study it with the new knowledge you have. Since the second argument to this function needs to be assigned a value (which it looks like it's assigned a pointer to a structure) you can't simply force this function to return 0. But, after looking at it a bit, you can begin to see that once the second argument is assigned it's initial values at the beginning, nothing much else is really done with it. There's one place where two things are assigned 0 at the end, but that also results in the return code being assigned -7300, so that's a BAD BOY anyway. So it would appear that once a2 is set up initially, the rest can be bypassed and a zero returned.

Since a2 is a pointer to a structure that is used several other places in the program, we would have to reverse exactly HOW it is used, and then we'd have to do all sorts of calculations and dig into assembly code to write our own version of the initialization. No, here, it's best to leave it as is, since once it's already initialized in this function, it seems to hold the correct answers already, as long as those two bytes

towards the end are not assigned to zero, because that leads to the BAD BOY value of -7300 being returned:



You can see v23 is assigned -7300, and then v23 is assigned to v6, and v6 is what's returned. And remember, the return code must be zero! Since the only thing we need from this function is for the structure assigned to a2 to be initialized, and it's initialization is done at the beginning of the function, and all the rest of the function is just deciding the return code, would it be possible to just do a

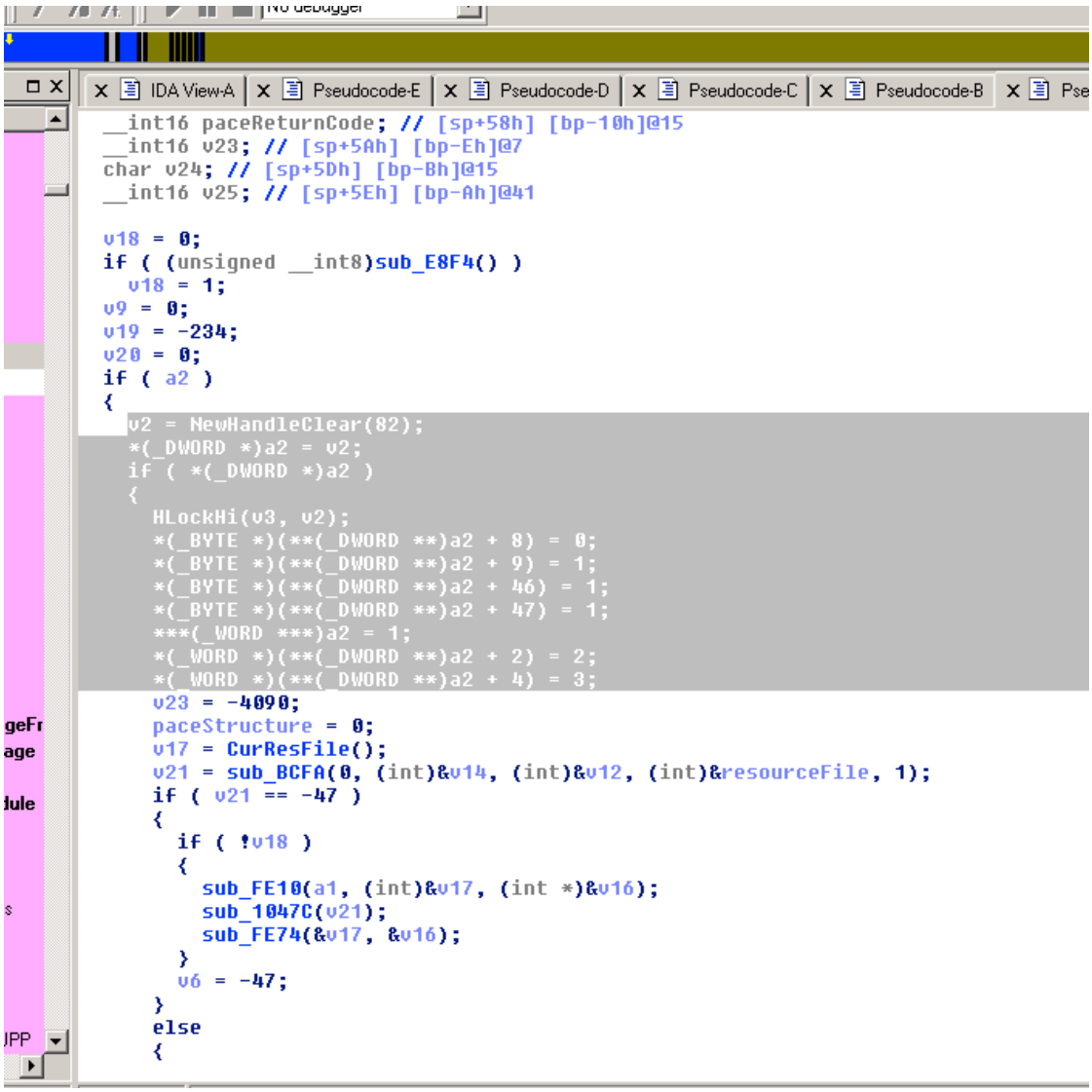
xor eax, eax
ret

right after a2 is fully initialized?

HINT: you can right-click to the right of one of the lines in IDA Pro and chose "Copy to Assembly", and it will copy the decompilation to the assembly view.

Additionally, if you go to Options->General menu, you can change the "number of opcode bytes" to 12, to get the opcodes used listed on the left.

Remember previously that we did indeed find other places in the program that interacted with a2 (or rather, unk_53D154). But so far, we've isolated the whole additional prot to this one function, and the structure that belongs in a2 (or rather, unk_53D154) and the value that is returned. This is what we know about the protection. This function must return 0, and a2 must be initialized. So let's take a look again at what happens to a2 in this function:



Here, a new 82-byte pointer is returned. This pointer is assigned to a2 (remember, unk_53D154), and then this pointer is initialized to some initial values. You can see this highlighted in gray. The only other place in this function where the values are changed are towards the bottom, where some values get assigned 0. BUT, when these values are changed, that only happens in a BAD BOY. So it's safe to say that for this function anyway, the only necessary part happens right here, where a2 is initialized.

Everything else is just calculating the return value, and we already know what we want it to be 😈
Now, a2 is initialized whether the rest of the function continues or not, so is the rest of the function even necessary? So, right-click next to V24 = -4090 and choose 'Copy to assembly'
the goto to assembly & select the part after V24 = -4090, which should select the same part in hex view.
use a hexeditor. So what happens when you try 31C0C9C3 for the patch instead? 😉

Reversing Waves Plugins V7r12

By unsanity MSJ

Step 1: Reconnaissance

or, ***What the Hell Am I Doing?***

Tools Required:

unpace

AudioUnitHosting.app (placed in your /Applications folder) found here:
http://www.mediafire.com/?kxwmxxwezao

symset, found here: http://www.mediafire.com/?kzeoopdoql2hy7m

Waves v7r12 Installer (which can be downloaded from the Waves website)

OTX and GDB or **IDA** (your choice, though IDA is much easier)

Additional tools required:

a brain

a pair of eyes

OK, first things first. Install Waves. During the install, check every single little checkbox. We want to install everything and crack it all in one fell swoop.

While that's installing, you can go ahead and place symset in your /usr/local/bin, /usr/bin, or wherever you like to place your little command-line utilities. Once the install is done, restart your Mac.

OK, we're back in our lovely little OS, now it's time to find out exactly what the Waves protection consists of. In /Applications, we see a new "Waves" folder, and in there we see some standalone apps, waveshells, and a whole folder full of plugins. For now, we'll just work on the plugins and the apps will come later. So it looks like the only thing that waves installs to the normal plugin locations (/Library/Audio/Plug-Ins/Components for AU for example) is the Waveshells. Hmm, let's try to unpace those:

Code:

```
69-69-69-199:~ Unsanity$ unpace -f /Library/Audio/Plug-
Ins/Components/WaveShell-AU\ 7.1.component

   PACE InterLok v5.4 - 5.7.2 unwrapper for Intel architecture
```

```
   TEAM XVX 2010

[!] Executable:   /Library/Audio/Plug-Ins/Components/WaveShell-AU
7.1.component/Contents/MacOS/WaveShell-AU
[!] Info:   this is a fat binary, thinning ...
[!] Info:   thinning done, continuing ...
[!] Info:   seems like it's not protected with InterLok, exiting ...
69-69-69-199:~ Unsanity$
```

Hmm ok, so the waveshells are not pace-protected, it must be in the individual plugins. Let's try to unpace one. Copy the API-2500.bundle to the desktop and then unpace it:

Code:

```
69-69-69-199:~ Unsanity$ unpace -f /Users/Unsanity/Desktop/API-2500.bundle

   PACE InterLok v5.4 - 5.7.2 unwrapper for Intel architecture
   TEAM XVX 2010

[!] Executable:   /Users/Unsanity/Desktop/API-2500.bundle/Contents/MacOS/API-
2500
[!] Version:   PACE InterLok v5.7.2, release date 2009-07-24



[!] Info:   child process started pid: 256, thread: 5379
[!] Info:   executing /Users/Unsanity/Desktop/API-
2500.bundle/Contents/MacOS/API-2500
[!] Info:   Loading the library ...
[!] Error:   unable to open /Users/Unsanity/Desktop/API-
2500.bundle/Contents/MacOS/API-2500:  dlopen(/Users/Unsanity/Desktop/API-
2500.bundle/Contents/MacOS/API-2500, 6): Library not loaded:
@loader_path/../../../WavesLib 7.1.framework/Versions/A/WavesLib
  Referenced from: /Users/Unsanity/Desktop/API-
2500.bundle/Contents/MacOS/API-2500
  Reason: image not found
[!] Info:   child process terminated
69-69-69-199:~ Unsanity$
```

Hmmm it's looking for a framework and can't find it. ../../../ refers to the same directory as the plugin. Ok, so leave the copy on the desktop in case we mess it up, and then unpace the original plug:

Code:

```
69-69-69-199:~ Unsanity$ /Applications/Waves/Waves Plug-Ins/API-
2500.bundle/Contents/MacOS/API-2500

   PACE InterLok v5.4 - 5.7.2 unwrapper for Intel architecture
   TEAM XVX 2010

[!] Executable:   /Applications/Waves/Waves Plug-Ins/API-
2500.bundle/Contents/MacOS/API-2500
[!] Version:   PACE InterLok v5.7.2, release date 2009-07-24


[!] Info:   child process started pid: 250, thread: 5379
[!] Info:   executing /Applications/Waves/Waves Plug-Ins/API-
2500.bundle/Contents/MacOS/API-2500
[!] Info:   Loading the library ...
```

and it just hangs there. Hmm. Perhaps the framework is protected and it's confusing unpace? Let's see what's all protected here:

Code:

```
69-69-69-199:~ Unsanity$ cd /Applications/Waves/Waves\ Plug-Ins
69-69-69-199:Waves Plug-Ins Unsanity$ grep -i -r sgemexam .
Binary file ./API-2500.bundle/Contents/MacOS/API-2500 matches
Binary file ./API-550.bundle/Contents/MacOS/API-550 matches
Binary file ./API-560.bundle/Contents/MacOS/API-560 matches
Binary file ./ArtistLib 7.1.framework/Versions/A/WavesLib matches
Binary file ./ArtistLib 7.1.framework/Versions/Current/WavesLib matches
Binary file ./ArtistLib 7.1.framework/WavesLib matches
Binary file ./AudioTrack.bundle/Contents/MacOS/AudioTrack matches
Binary file ./C1.bundle/Contents/MacOS/C1 matches
Binary file ./C360.bundle/Contents/MacOS/C360 matches
Binary file ./C4.bundle/Contents/MacOS/C4 matches
```

```
Binary file ./Center.bundle/Contents/MacOS/Center matches
Binary file ./CLA Bass.bundle/Contents/MacOS/CLA Bass matches
Binary file ./CLA Drums.bundle/Contents/MacOS/CLA Drums matches
Binary file ./CLA Effects.bundle/Contents/MacOS/CLA Effects matches
Binary file ./CLA Guitars.bundle/Contents/MacOS/CLA Guitars matches
Binary file ./CLA Unplugged.bundle/Contents/MacOS/CLA Unplugged matches
Binary file ./CLA Vocals.bundle/Contents/MacOS/CLA Vocals matches
Binary file ./CLA-2A.bundle/Contents/MacOS/CLA-2A matches
Binary file ./CLA-3A.bundle/Contents/MacOS/CLA-3A matches
Binary file ./CLA-76.bundle/Contents/MacOS/CLA-76 matches
Binary file ./DeBreath.bundle/Contents/MacOS/DeBreath matches
Binary file ./DeEsser.bundle/Contents/MacOS/DeEsser matches
Binary file ./DLA.bundle/Contents/MacOS/DLA matches
Binary file ./Doppler.bundle/Contents/MacOS/Doppler matches
Binary file ./Dorrough Surround 5.0.bundle/Contents/MacOS/Dorrough Surround 5.0 matches
Binary file ./Dorrough Surround 5.1.bundle/Contents/MacOS/Dorrough Surround 5.1 matches
Binary file ./Dorrough.bundle/Contents/MacOS/Dorrough matches
Binary file ./Doubler.bundle/Contents/MacOS/Doubler matches
Binary file ./EddieKramer BA.bundle/Contents/MacOS/EddieKramer BA matches
Binary file ./EddieKramer DR.bundle/Contents/MacOS/EddieKramer DR matches
Binary file ./EddieKramer FX.bundle/Contents/MacOS/EddieKramer FX matches
Binary file ./EddieKramer GT.bundle/Contents/MacOS/EddieKramer GT matches
Binary file ./EddieKramer VC.bundle/Contents/MacOS/EddieKramer VC matches
Binary file ./Enigma.bundle/Contents/MacOS/Enigma matches
Binary file ./GTRAmp.bundle/Contents/MacOS/GTRAmp matches
Binary file ./GTRSoloLib 7.1.framework/Versions/A/WavesLib matches
Binary file ./GTRSoloLib 7.1.framework/Versions/Current/WavesLib matches
Binary file ./GTRSoloLib 7.1.framework/WavesLib matches
Binary file ./GTRSoloToolRack.bundle/Contents/MacOS/GTRSoloToolRack matches
Binary file ./GTRStomp.bundle/Contents/MacOS/GTRStomp matches
Binary file ./GTRToolRack.bundle/Contents/MacOS/GTRToolRack matches
Binary file ./GTRTuner.bundle/Contents/MacOS/GTRTuner matches
Binary file ./H-Comp.bundle/Contents/MacOS/H-Comp matches
Binary file ./H-Delay.bundle/Contents/MacOS/H-Delay matches
Binary file ./IDR.bundle/Contents/MacOS/IDR matches
Binary file ./IR-1.bundle/Contents/MacOS/IR-1 matches
Binary file ./IR-360.bundle/Contents/MacOS/IR-360 matches
Binary file ./IR-L.bundle/Contents/MacOS/IR-L matches
```

```
Binary file ./JJP-Bass.bundle/Contents/MacOS/JJP-Bass matches
Binary file ./JJP-Cymb-Perc.bundle/Contents/MacOS/JJP-Cymb-Perc matches
Binary file ./JJP-Drums.bundle/Contents/MacOS/JJP-Drums matches
Binary file ./JJP-Guitars.bundle/Contents/MacOS/JJP-Guitars matches
Binary file ./JJP-Strings-Keys.bundle/Contents/MacOS/JJP-Strings-Keys matches
Binary file ./JJP-Vocals.bundle/Contents/MacOS/JJP-Vocals matches
Binary file ./KramerHLS.bundle/Contents/MacOS/KramerHLS matches
Binary file ./KramerPIE.bundle/Contents/MacOS/KramerPIE matches
Binary file ./L1.bundle/Contents/MacOS/L1 matches
Binary file ./L2.bundle/Contents/MacOS/L2 matches
Binary file ./L3 Multi.bundle/Contents/MacOS/L3 Multi matches
Binary file ./L3 Ultra.bundle/Contents/MacOS/L3 Ultra matches
Binary file ./L3-16.bundle/Contents/MacOS/L3-16 matches
Binary file ./L3-LL Multi.bundle/Contents/MacOS/L3-LL Multi matches
Binary file ./L3-LL Ultra.bundle/Contents/MacOS/L3-LL Ultra matches
Binary file ./L360.bundle/Contents/MacOS/L360 matches
Binary file ./LFE360.bundle/Contents/MacOS/LFE360 matches
Binary file ./LinEQ.bundle/Contents/MacOS/LinEQ matches
Binary file ./LinMB.bundle/Contents/MacOS/LinMB matches
Binary file ./LoAir.bundle/Contents/MacOS/LoAir matches
Binary file ./Maserati ACG.bundle/Contents/MacOS/Maserati ACG matches
Binary file ./Maserati B72.bundle/Contents/MacOS/Maserati B72 matches
Binary file ./Maserati DRM.bundle/Contents/MacOS/Maserati DRM matches
Binary file ./Maserati GTi.bundle/Contents/MacOS/Maserati GTi matches
Binary file ./Maserati HMX.bundle/Contents/MacOS/Maserati HMX matches
Binary file ./Maserati VX1.bundle/Contents/MacOS/Maserati VX1 matches
Binary file ./MaxxBass.bundle/Contents/MacOS/MaxxBass matches
Binary file ./MaxxVolume.bundle/Contents/MacOS/MaxxVolume matches
Binary file ./MetaFlanger.bundle/Contents/MacOS/MetaFlanger matches
Binary file ./MondoMod.bundle/Contents/MacOS/MondoMod matches
Binary file ./Morphoder.bundle/Contents/MacOS/Morphoder matches
Binary file ./MultiRack.bundle/Contents/MacOS/MultiRack matches
Binary file ./MV2.bundle/Contents/MacOS/MV2 matches
Binary file ./MV360.bundle/Contents/MacOS/MV360 matches
Binary file ./PAZ.bundle/Contents/MacOS/PAZ matches
Binary file ./PS22.bundle/Contents/MacOS/PS22 matches
Binary file ./PuigChild.bundle/Contents/MacOS/PuigChild matches
Binary file ./PuigTec.bundle/Contents/MacOS/PuigTec matches
```

```
Binary file ./Q-Clone.bundle/Contents/MacOS/Q-Clone matches
Binary file ./Q10.bundle/Contents/MacOS/Q10 matches
Binary file ./R360.bundle/Contents/MacOS/R360 matches
Binary file ./RBass.bundle/Contents/MacOS/RBass matches
Binary file ./RChannel.bundle/Contents/MacOS/RChannel matches
Binary file ./RComp.bundle/Contents/MacOS/RComp matches
Binary file ./RDeEsser.bundle/Contents/MacOS/RDeEsser matches
Binary file ./RenAxx.bundle/Contents/MacOS/RenAxx matches
Binary file ./REQ.bundle/Contents/MacOS/REQ matches
Binary file ./RVerb.bundle/Contents/MacOS/RVerb matches
Binary file ./RVox.bundle/Contents/MacOS/RVox matches
Binary file ./S1.bundle/Contents/MacOS/S1 matches
Binary file ./S360.bundle/Contents/MacOS/S360 matches
Binary file ./SoundShifter.bundle/Contents/MacOS/SoundShifter matches
Binary file ./SSLChannel.bundle/Contents/MacOS/SSLChannel matches
Binary file ./SSLComp.bundle/Contents/MacOS/SSLComp matches
Binary file ./SSLEQ.bundle/Contents/MacOS/SSLEQ matches
Binary file ./StompsLib 7.1.framework/Versions/A/WavesLib matches
Binary file ./StompsLib 7.1.framework/Versions/Current/WavesLib matches
Binary file ./StompsLib 7.1.framework/WavesLib matches
Binary file ./SuperTap.bundle/Contents/MacOS/SuperTap matches
Binary file ./TransX.bundle/Contents/MacOS/TransX matches
Binary file ./TrueVerb.bundle/Contents/MacOS/TrueVerb matches
Binary file ./UltraPitch.bundle/Contents/MacOS/UltraPitch matches
Binary file ./UM.bundle/Contents/MacOS/UM matches
Binary file ./VComp.bundle/Contents/MacOS/VComp matches
Binary file ./VEQ3.bundle/Contents/MacOS/VEQ3 matches
Binary file ./VEQ4.bundle/Contents/MacOS/VEQ4 matches
Binary file ./Vocal Rider.bundle/Contents/MacOS/Vocal Rider matches
Binary file ./W43.bundle/Contents/MacOS/W43 matches
Binary file ./WavesTune LT.bundle/Contents/MacOS/WavesTune LT matches
Binary file ./WavesTune.bundle/Contents/MacOS/WavesTune matches
Binary file ./WNS.bundle/Contents/MacOS/WNS matches
Binary file ./X-Click.bundle/Contents/MacOS/X-Click matches
Binary file ./X-Crackle.bundle/Contents/MacOS/X-Crackle matches
Binary file ./X-Hum.bundle/Contents/MacOS/X-Hum matches
Binary file ./X-Noise.bundle/Contents/MacOS/X-Noise matches
```

```
Binary file ./Z-Noise.bundle/Contents/MacOS/Z-Noise matches
69-69-69-199:Waves Plug-Ins Unsanity$
```

OK, so we can see that indeed the frameworks are PACE-protected as well, and if a plugin loads the framework, it may be interfering with unpace. So let's go ahead and unpace those frameworks:

Code:

```
69-69-69-199:Waves Plug-Ins Unsanity$ for i in *.framework; do unpace -f
"$i"; done
```

The key to understanding this command is this:

the "for" sets up what files we'll be using; it loops over them. Here, we set "i" to be the variable that holds each filename, and use a "" as a wildcard, basically saying "all files" or "all files that end with .bundle". the "i" becomes the filename. We access the i variable by using the dollar sign, and we put quotes around it in case there are spaces in the filename. The "done" command signifies the end of the loop. the "do" command is the command we want to run in the loop.*

So if we remember back to what XVX said about Waves, they mentioned that the auhost flag must be used, and I can confirm that indeed it does. So now that we've got the frameworks out of the way so they won't interfere with plug-in loading, let's unpace the plugins. There is a very specific way that this must be done, and if you click something wrong throughout the whole process, you will render the plug-ins useless. So now's a good time to make a backup of the waves plug-ins folder just in case.

So here's the command we're going to use to unpace: (make sure you're in the Waves Plug-Ins folder)

Code:

```
for i in *.bundle; do unpace -f "$i" -h auhost; done
```

When you do this, this is what you'll see in terminal:

Code:

```
69-69-69-199:Waves Plug-Ins Unsanity$ for i in *.bundle; do unpace -h auhost
-f "$i"; done

   PACE InterLok v5.4 - 5.7.2 unwrapper for Intel architecture
   TEAM XVX 2010

[!] Executable:   /Applications/Waves/Waves Plug-Ins/API-
2500.bundle/Contents/MacOS/API-2500
```

```
[!] Version:   PACE InterLok v5.7.2, release date 2009-07-24


[!] Info:   child process started pid: 1476, thread: 5379
[!] Info:   executing /Applications/Waves/Waves Plug-Ins/API-
2500.bundle/Contents/MacOS/API-2500
[!] Info:   executing auhost...
```

The important part to notice here is what unpace says it's executing, in this example it's **/Applications/Waves/Waves Plug-Ins/API-2500.bundle/Contents/MacOS/API-2500**. So in AudioUnitHosting, we need to select the plugin from the list that corresponds to api-2500. There are 2 in the list, "Waves: API-2500 (m)" and "Waves: API-2500 (s)". There are both the same plugin, one is just for mono and one for stereo. It does not matter which one you choose, just as long as you choose the API-2500 one. Some of these names will be confusing, but before you select one from the list, be sure that it's the correct one. ALWAYS look at the terminal to see what unpace says it's currently executing and only choose that one! Also, be sure that the PACE window that comes up has the title of the one you're unpacing as well.

OK, go through the whole shebang and unpace each and every one of them. I'll see you back here in a bit

BTW, there are some that won't unpace, ps22.bundle, multirack.bundle, etc and maybe a few others. when you come to these they won't even show up in the list of plugins to choose from, just press ctrl+c to exit the unpace loop, move those plugins out of there to another folder, and re-run the unpace command.

EDIT: The TDM only plugins (like DLA, PS22, and others) will NOT show up in AudioUnitHosting list, and therefore cannot be unpaced this way. The only way to unpace these is to run unpace with a custom host (Pro Tools for example) that is TDM compatible (which means you must ahve TDM hardware) and then choose the plugin in Pro Tools. So when you find one that unpace is currently on and is not showing in the auhost list, stop unpace with ctrl+c, move the plugin out of the folder, and restart the unpace command.

Dorrough Surround 5.0 = Dorrough360 (5->5)
Dorrough Surround 5.1 = Dorrough360 (6->6)

L3 Multi is different than L3-LL Multi . On the PACE window that comes up while unpacing, NEVER

press the "cancel" button. ONLY press the "quit" button. If there's not quit button, press the "back" button, and then press the "quit" button.

Step 2: Analysis

or, ***What the Hell Do I Do Now?***

Additional tools required:
a brain

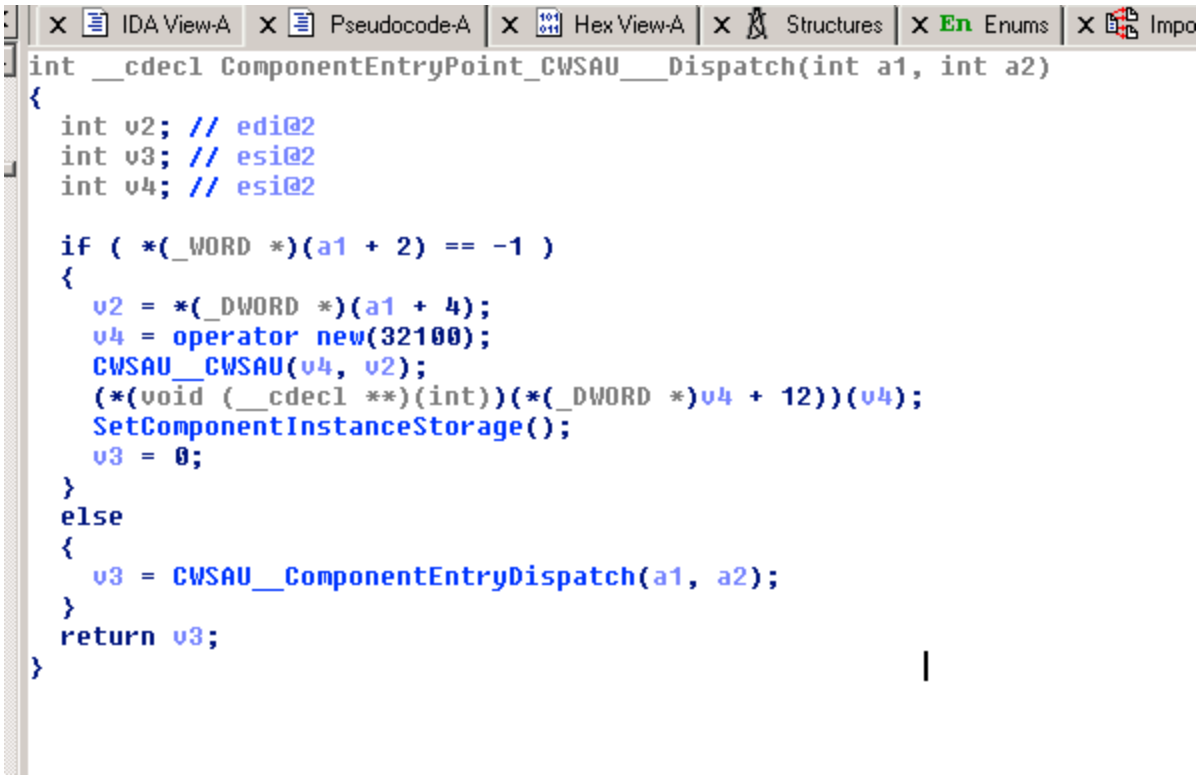
OK, so here comes the difficult part. These are a little harder to find where the protection lives at. If you run the AU Reg Utility, and then run AU Lab under gdb and try to break on IL_SC_D_PPC, you'll notice that there is no break, and the error you get back in the terminal from each plugin is -10875, which is not an authorization error, but a "component failed to open" error. So there is something going on to begin with. There is probably some protection in the Waveshell, so we'll begin there. I like to start with the Audio Units because I know more about how they're loaded, what functions they call, and how they implement copy protection. So after you've done plugins for a while, you'll know that for vst's the entrypoint is almost always VSTPluginMain, and for Audio Units it's always some form of blah blah Entry and blah blah ViewEntry. So I usually do a grep of the symbols like so:

Code:

```
nm -arch i386 /Applications/Waves/Waveshells/WaveShell-AU\
7.1.component/Contents/MacOS/WaveShell-AU | grep -i entry
```

Aha, top 2 results:
0001b4e6 T _CWSAUEntry
0001b4f0 T _CWSAUViewEntry

So now we can view it in IDA:

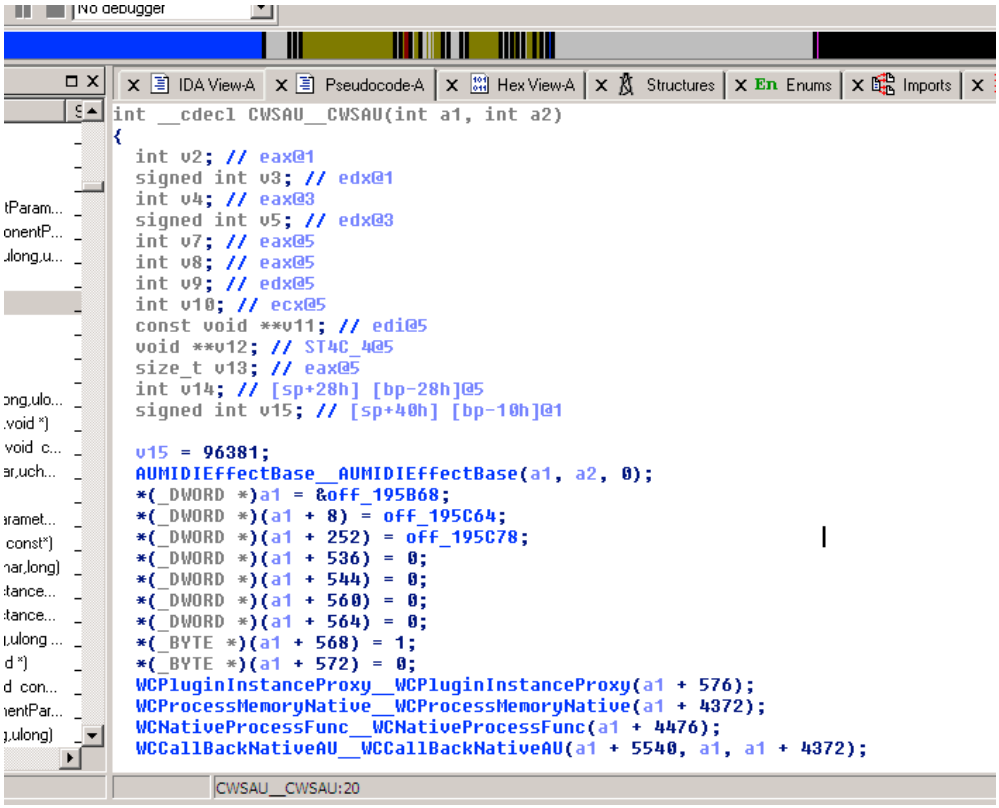


Now from here it's always the same in Audio Units... the entrypoint will allocate the class, and then call SetComponentInstanceStorage. So this is usual stuff. But notice the line that says:
(*(void (__cdecl **)(int))(*(_DWORD *)v4 + 12))(v4);
How can we tell where that goes??? Easy. We need to look at the vtable of the class that was allocated. So here is where the operator new was called, and the CWSAU::CWSAU is the constructor for the memory that was allocated.

v4 = operator new(32100);
CWSAU__CWSAU(v4, v2);

This is equivalent to this c++ code:
CWSAU *audioUnit = new CWSAU(v2);

So this is the constructor. If we follow the lin CWSAU::CWSAU in by double-clicking on it, we'll get to here:



Now remember that this a1 is the pointer that was passed in, and is literally the pointer to the class. See the line that says:

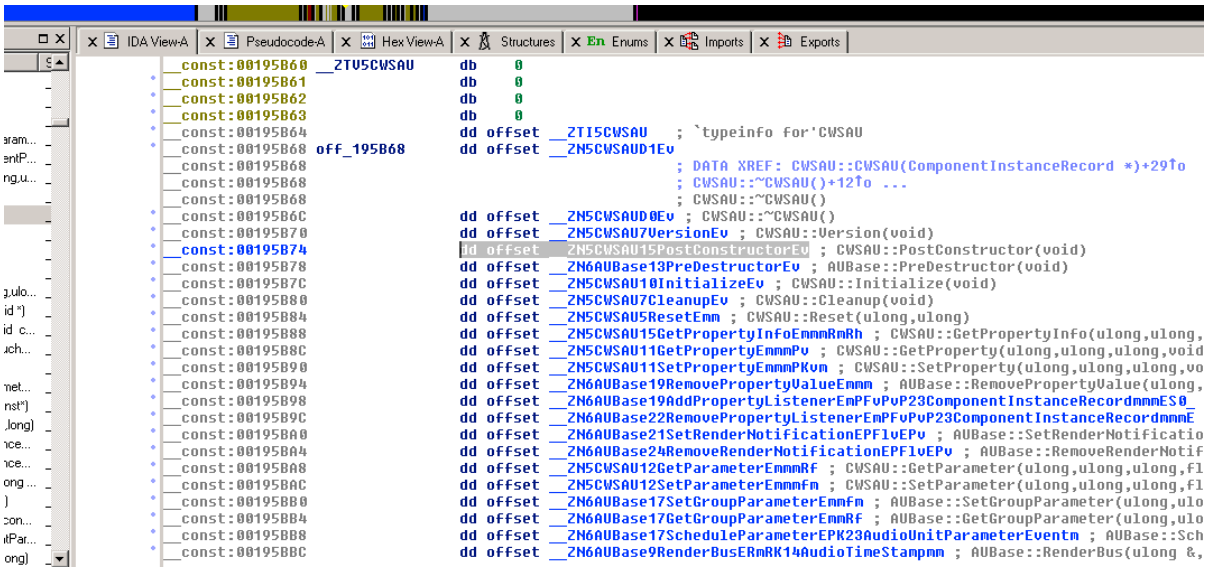
*(_DWORD *)a1 = &off_195B68;

This is the vtable, because the first thing in a c++ class is ALWAYS the v-table. This is an array of function pointers and is how subclassing and overloading is accomplished in c++. So we double-click on the off_195B68 and it takes us to the vtable 🙂

Now remember the line we were looking at

(*(void (__cdecl **)(int))(*(_DWORD *)v4 + 12))(v4);

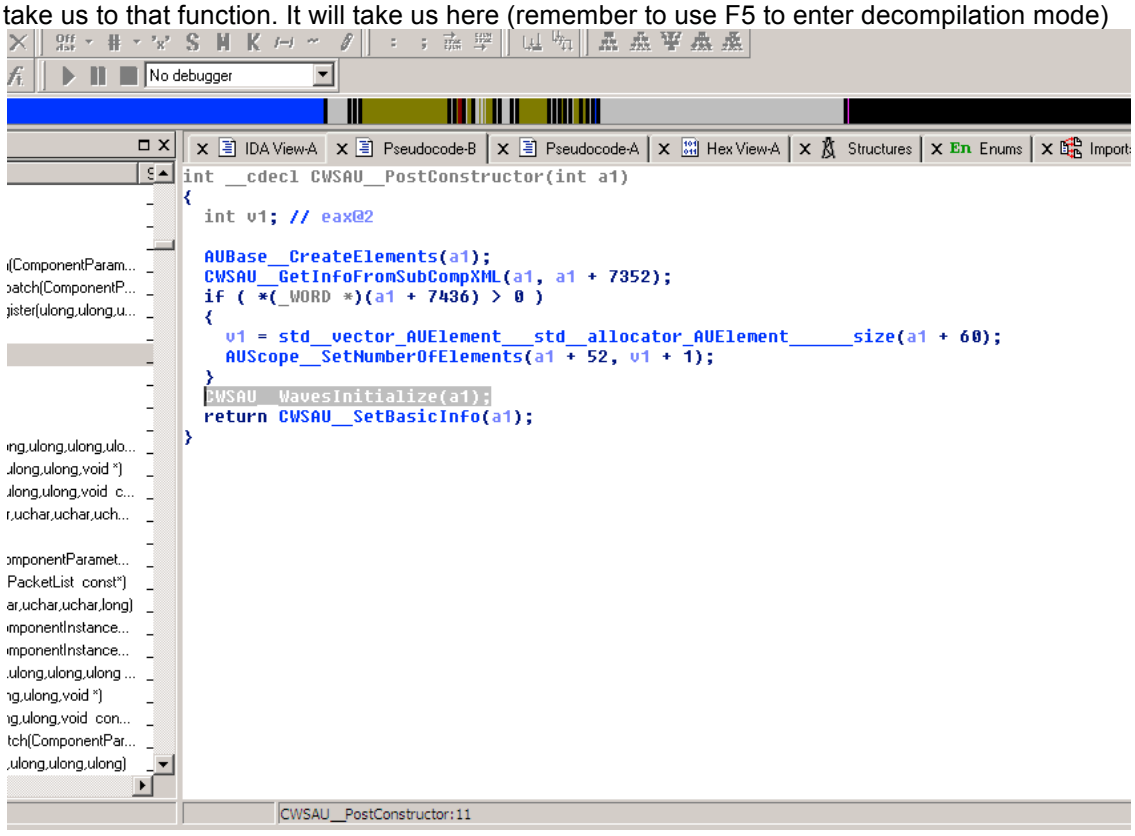
here v4 was the class pointer, it's dereferencing v4 (therefore getting the vtable pointer) and then adding 12 to it. So in this function list


IDA View-A
Pseudocode-A
Hex View-A
Structures
Enums
Imports
Exports
__const:00195B60 __ZTV5CWSAU db 0
__const:00195B61 db 0
__const:00195B62 db 0
__const:00195B63 db 0
__const:00195B64 dd offset __ZTI5CWSAU ; `typeinfo for'CWSAU
__const:00195B68 off_195B68 dd offset __ZN5CWSAUD1Ev
__const:00195B68 ; DATA XREF: CWSAU::CWSAU(ComponentInstanceRecord *)+29↑o
__const:00195B68 ; CWSAU::~CWSAU()+12↑o ...
__const:00195B68 ; CWSAU::~CWSAU()
__const:00195B6C dd offset __ZN5CWSAUD0Ev ; CWSAU::~CWSAU()
__const:00195B70 dd offset __ZN5CWSAU7VersionEv ; CWSAU::Version(void)
__const:00195B74 dd offset __ZN5CWSAU15PostConstructorEv ; CWSAU::PostConstructor(void)
__const:00195B78 dd offset __ZN6AUBase13PreDestructorEv ; AUBase::PreDestructor(void)
__const:00195B7C dd offset __ZN5CWSAU10InitializeEv ; CWSAU::Initialize(void)
__const:00195B80 dd offset __ZN5CWSAU7CleanupEv ; CWSAU::Cleanup(void)
__const:00195B84 dd offset __ZN5CWSAU5ResetEmm ; CWSAU::Reset(ulong,ulong)
__const:00195B88 dd offset __ZN5CWSAU15GetPropertyInfoEmmmRmRh ; CWSAU::GetPropertyInfo(ulong,ulong,
__const:00195B8C dd offset __ZN5CWSAU11GetPropertyEmmmPv ; CWSAU::GetProperty(ulong,ulong,ulong,void
__const:00195B90 dd offset __ZN5CWSAU11SetPropertyEmmmPKvm ; CWSAU::SetProperty(ulong,ulong,ulong,vo
__const:00195B94 dd offset __ZN6AUBase19RemovePropertyValueEmmm ; AUBase::RemovePropertyValue(ulong,
__const:00195B98 dd offset __ZN6AUBase19AddPropertyListenerEmPFvPvP23ComponentInstanceRecordmmmES0_
__const:00195B9C dd offset __ZN6AUBase22RemovePropertyListenerEmPFvPvP23ComponentInstanceRecordmmmE
__const:00195BA0 dd offset __ZN6AUBase21SetRenderNotificationEPFlvEPv ; AUBase::SetRenderNotificatio
__const:00195BA4 dd offset __ZN6AUBase24RemoveRenderNotificationEPFlvEPv ; AUBase::RemoveRenderNotif
__const:00195BA8 dd offset __ZN5CWSAU12GetParameterEmmmRf ; CWSAU::GetParameter(ulong,ulong,ulong,fl
__const:00195BAC dd offset __ZN5CWSAU12SetParameterEmmmfm ; CWSAU::SetParameter(ulong,ulong,ulong,fl
__const:00195BB0 dd offset __ZN6AUBase17SetGroupParameterEmmfm ; AUBase::SetGroupParameter(ulong,ulo
__const:00195BB4 dd offset __ZN6AUBase17GetGroupParameterEmmRf ; AUBase::GetGroupParameter(ulong,ulo
__const:00195BB8 dd offset __ZN6AUBase17ScheduleParameterEPK23AudioUnitParameterEventm ; AUBase::Sch
__const:00195BBC dd offset __ZN6AUBase9RenderBusERmRK14AudioTimeStampmm ; AUBase::RenderBus(ulong &,

take note of where IDA put your cursor when you double-clicked on off_195B68, and then go ahead 12 (each line is 4 bytes, so you'll move ahead by increments of 4, for a total of 3 lines). I have it highlighted in the screenshot. So PostConstructor is what is being called in the line

(*(void (__cdecl **)(int))(*(_DWORD *)v4 + 12))(v4);

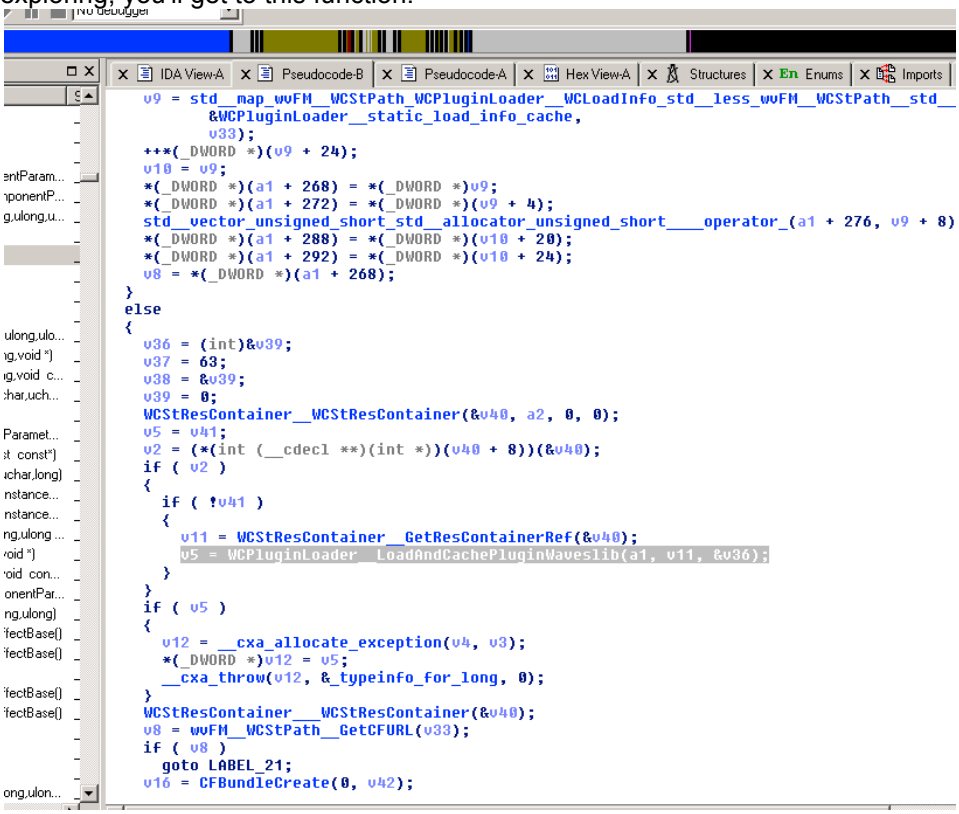
So in this function list, we can go ahead and double-click on the line I have highlighted there and it will take us to that function. It will take us here (remember to use F5 to enter decompilation mode)



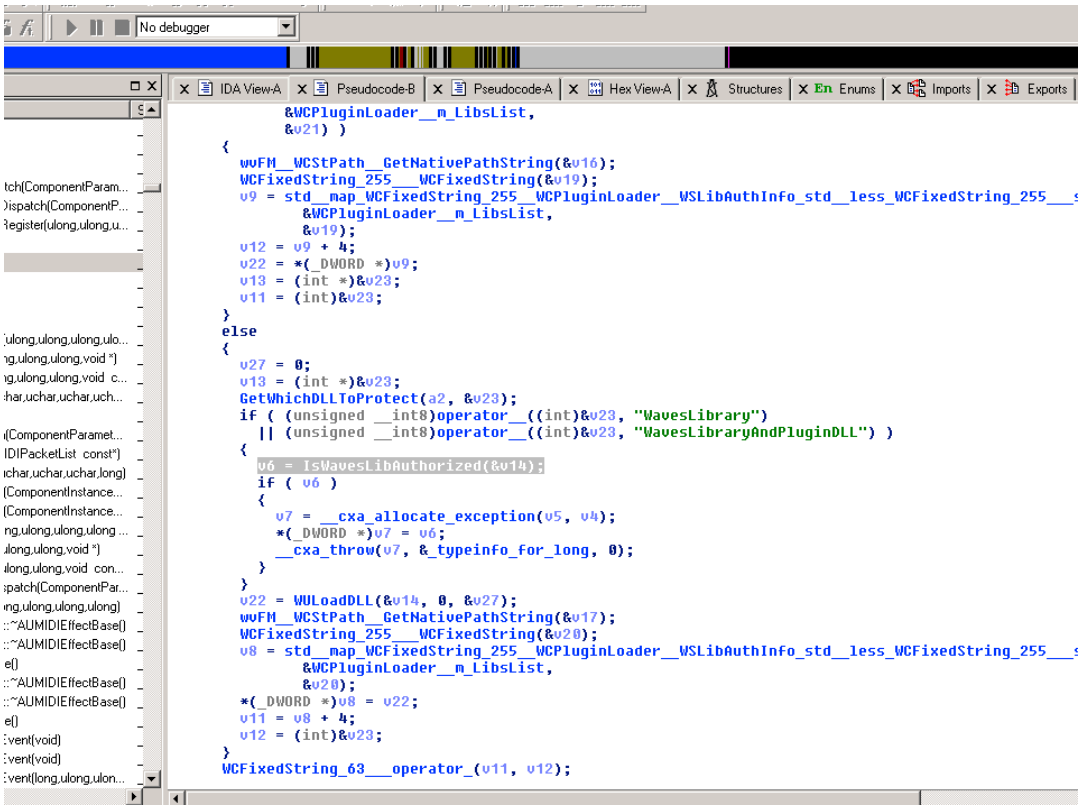
I have highlighted the interesting function, WavesInitialize()... hmm, wonder if that has to do with the protection

So in this function there's a mass of functions called. Some look interesting (like PluginLoader, and LoadPluginFromFile or whatever) so just follow those and take a look at what they're doing. Hmm mdoesn't seem to have anything to do with protection. So we'll peek in StartAUShell.,. AHA, in here we see some interesting stuff, like LoadPluginDLL, so we'll follow those in. So far it looks like once the AU is instantiated, it runs some post-constructor stuff, which includes loading the plugin we selected from the list in the host...

Inside LoadPluginDLL, we see WCPluginLoader__LoadPlugin, so we follow that in. OK, the goal here is to understand what's happening during the loading of the plugin, so take some time, explore the functions, make comments in IDA, rename functions and variables so it's easier to read. After some exploring, you'll get to this function:

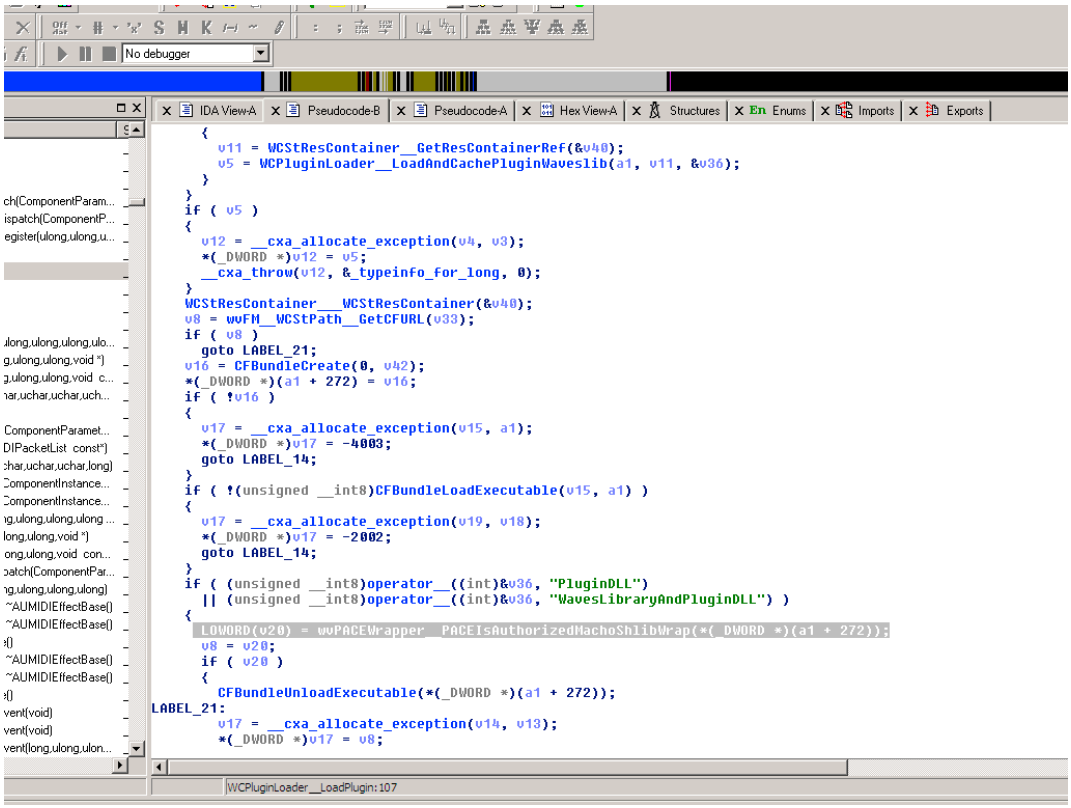


And finally, inside there, you'll find a function called "IsWavesLibAuthorized" that makes some PACE calls!



You'll notice that it returns something into v6, and right after there's a check that if it's true, it throws an exception. AHA! So we need to make that function return 0

OK, so we back up a little bit and keep exploring. Then we'll see this here:



Once you've explored the highlighted function a bit, you'll know that it needs to return 0 as well, otherwise CFBundleUnloadExecutable is called, adn we don't want that do we?

OK, so there doesn't seem to be much else going on in here, so we'll try those two patches in the waveshells and give it a go!

Here's a quick hint on an easy way to patch the waveshells, which is what I use symset for.
Here's the usage screen for symset

Code:

```
   Symset - search for a symbol in binary and set a return value for it
   TEAM XVX 2010

   usage:    symset -f <filename> -a <arch> -s <symbol name> -r <return
value> [-q]
      -f, --file   path to the binary to search
      -a, --arch   architecture to search in (i386 or x86_64)
      -s, --symbol   mangled name of the symbol (include leading underscores)
      -r, --retvalue   the value to set the symbol to immediately return
      -q, --quiet   quiet output
```

So using that, we see we need to get the symbol names that we want to patch. We know they're called IsWavesLibAuthorized and PACEIsAuthorizedMachoShlibWrap and we have to set those to 0... let's get the actual symbol names...

Code:

```
69-69-69-199:~ Unsanity$ nm -arch i386
/Applications/Waves/Waveshells/WaveShell-AU\
7.1.component/Contents/MacOS/WaveShell-AU | grep -i iswaveslibauthorized
001845f8 s __GLOBAL__I__Z20IsWavesLibAuthorizedRN4wvFM8WCStPathE
0003e276 t __Z20IsWavesLibAuthorizedRN4wvFM8WCStPathE
69-69-69-199:~ Unsanity$
```

we don't want the __GLOBAL__ symbol, we want the one in the __text section, which you can tell by the little "t" next to the address. So the symbol name for that is __Z20IsWavesLibAuthorizedRN4wvFM8WCStPathE.

OK, so we can set this in symset by using the following command:

Code:

```
symset --file=/Applications/Waves/Waveshells/WaveShell-AU\
7.1.component/Contents/MacOS/WaveShell-AU --arch=i386 --
symbol=__Z20IsWavesLibAuthorizedRN4wvFM8WCStPathE --retvalue=0
```

Bingo! We get success! We can use this same technique to find the other symbol as well.

Code:

```
nm -arch i386 /Applications/Waves/Waveshells/WaveShell-AU\
7.1.component/Contents/MacOS/WaveShell-AU | grep -i
PACEIsAuthorizedMachoShlibWrap
0004146e t __ZN13wvPACEWrapper30PACEIsAuthorizedMachoShlibWrapEP10__CFBundle
```

so the symbol for that one is __ZN13wvPACEWrapper30PACEIsAuthorizedMachoShlibWrapEP10__CFBundle.

Code:

```
symset --file=/Applications/Waves/Waveshells/WaveShell-AU\
7.1.component/Contents/MacOS/WaveShell-AU --arch=i386 --
symbol=__ZN13wvPACEWrapper30PACEIsAuthorizedMachoShlibWrapEP10__CFBundle --
retvalue=0

   Symset - search for a symbol in binary and set a return value for it
   TEAM XVX 2010
```

```
Looking up symbol
__ZN13wvPACEWrapper30PACEIsAuthorizedMachoShlibWrapEP10__CFBundle in file
Found symbol!
Setting return value...
success
```

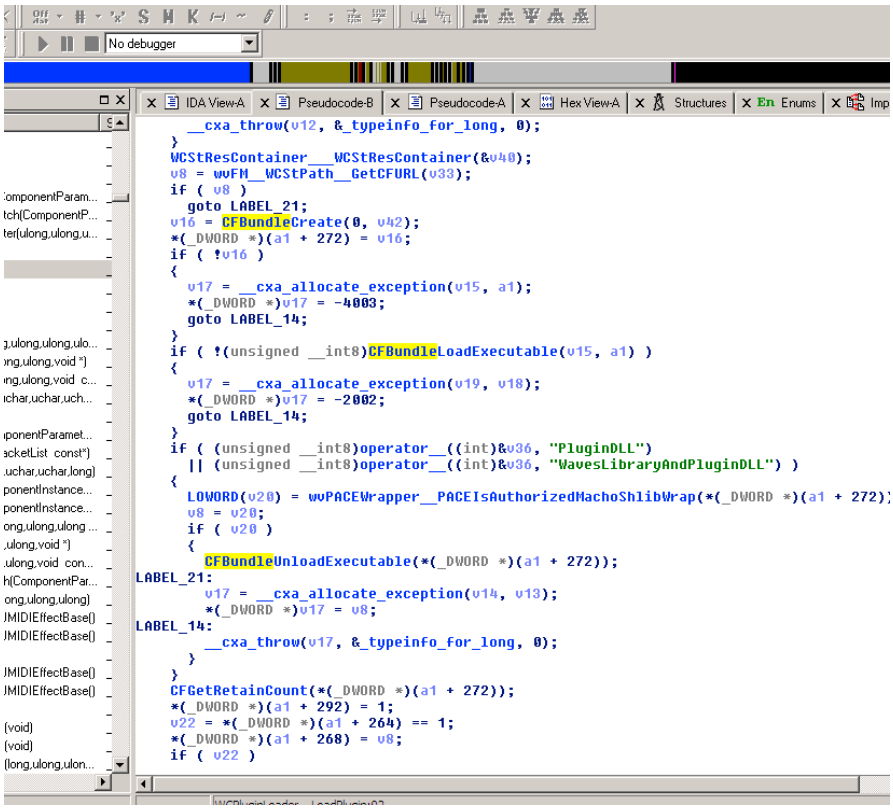
Step 3: Finishing Up

OK, If you've followed me up until now, you've probably noticed that the plugins do not load in a DAW yet. So at this point, we don't know whether we've made all the patches in the waveshell we needed to or not. So we need to find out if the plugin is being loaded or not. The best way to do that is simple: set a breakpoint on one of the bundle's entrypoints, and if we break, then we can go from there.

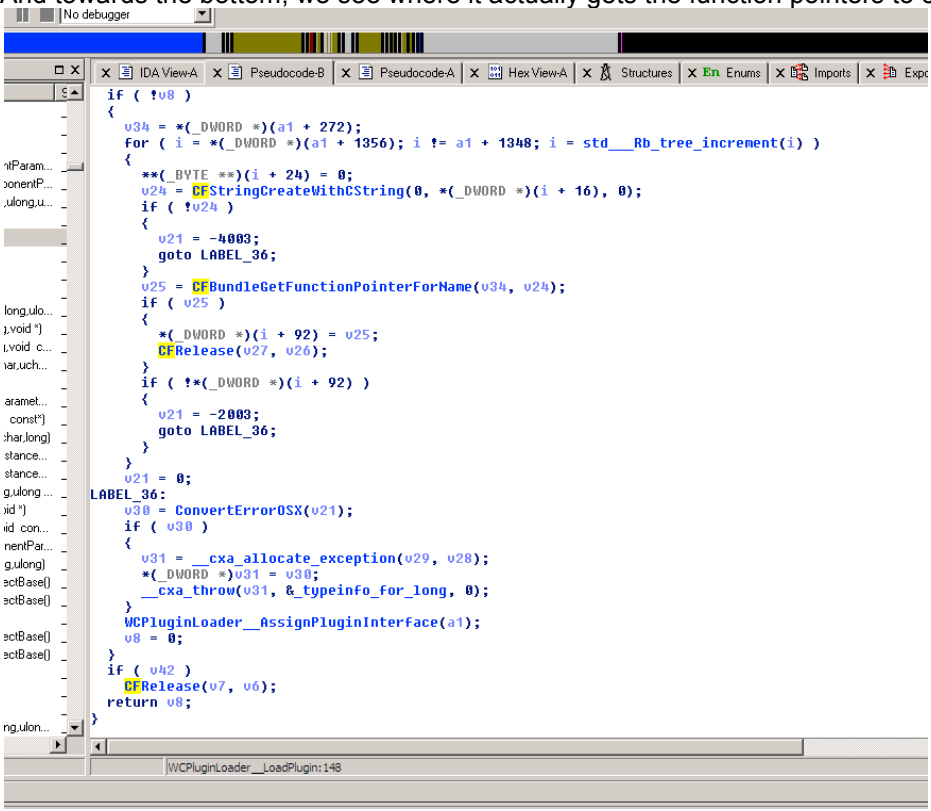
But how do we find the bundle's entrypoint?

The process is similar for this particular protection, as well as just in general finding out what gets called first from external bundles. We know that the waveshell is what is loaded by the DAW, and somehow the waveshell passes execution to the correct bundle... so that means that there is code somewhere in the waveshell that determines what function in what bundle to call. The first place that comes to mind to start checking is the code that we've already seen in the waveshell that loads and checks the plugin. So, if you need to, use step 2 to follow from CWSAUEntry up to where CWSAU_StartAUShell get's called here:

Follow that in and begin looking at the plugin loading code by following **WCPluginInstanceProxy__LoadPluginDLL** in, and then **WCPluginLoader__LoadPlugin**. OK, in this function, we see some of what I mentioned in the hint. Looks like the waveshell is using the CoreFoundation api's to load the bundles, as we can see here:



And towards the bottom, we see where it actually gets the function pointers to call:



CFBundleGetFunctionPointerForName(CFBundleRef bundle, char *name)... so if we were to break right on this function call in gdb, and then check out the second argument, we'd see what function names are getting saved. As it is, it's very hard to tell from the static code what's going on because of the use of a std::map and instance variables, in addition to a loop that seems to get multiple functions. But wait, before we fire up gdb to check, let's look at the last function there, WCPluginLoader__AssignPluginInterface....

Well, these look like plausible function names. Let's check the first one real quick, using any random bundle...

Code:

```
nm -arch i386 /Applications/Waves/Waves\ Plug-Ins/API-
2500.bundle/Contents/MacOS/API-2500 | grep -i wp_createinstance
```

Basically, with this command, we are running nm to get all the symbol names from the binary, and we pass it through grep -i to *case-insensitively* filter out the string "wp_createinstance", which is the first string we saw in the AssignPluginInterface function in IDA. Sure enough, after that command, we get back:

0000d1fe T _WP_CreateInstance

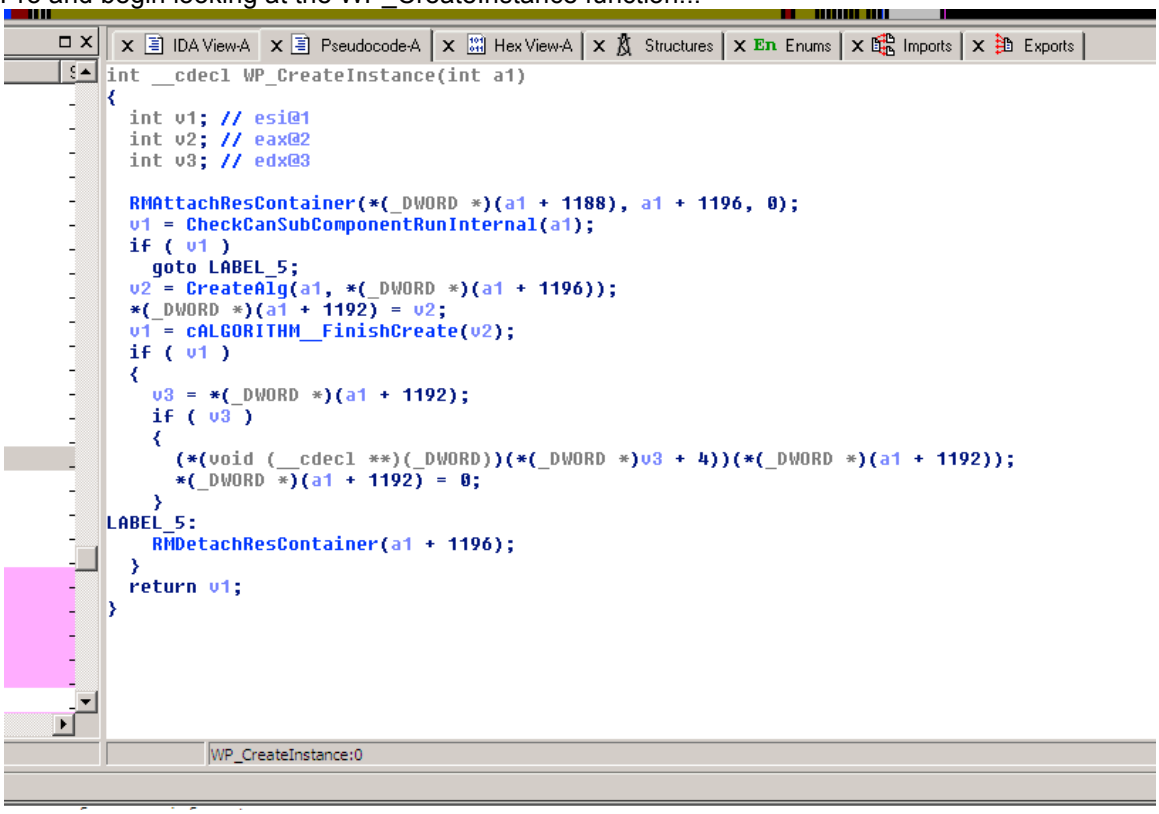
which shows that indeed it is a function in the bundle. So it looks like these strings in AssignPluginInterface are function names that are possible getting cached. Now let's look for one here that may be an entrypoint... WP_InitAlg looks plausible, but actually WP_CreateInstance sounds most like an entrypoint. Let's test it by running AU Lab and breaking on it... this will tell us if we are making it past the waveshell too.

Code:

```
gdb /Developer/Applications/Audio/AU\ Lab.app/Contents/MacOS/i386_AU_Lab
source /.gdbinit
hb WP_CreateInstance
```

At this point, you'll get a future break on it...
Function "WP_CreateInstance" not defined.
Hardware assisted breakpoint 1 (WP_CreateInstance) pending.

Go ahead and run gdb now with "r". AHA! We break in gdb before any of the authorization errors are spitted out! Seems we have found the entrypoint, or at least found a function that's called in the plugin before the authorization decision is made! So let's quit gdb for now, and load one of the bundles up in IDA Pro and begin looking at the WP_CreateInstance function...



Wow, first thing we see is a CheckCanSubComponentRunInternal function... which returns a value into v1. Then immediately afterward, we see that if v1 is not equal to 0, we jump to label_5, which basically skips over everything. Hmmm so it seems that we need to force CheckCanSubComponentRunInternal to always return 0! If we check a few of the plugins, we'll see that all of them have this function CheckCanSubComponentRunInternal, and we have to patch it in each plugin.... That's a lot of patching to do by hand. Good thing with symset we can automate it. Let's see what the actual symbol name is:

Code:

```
nm -arch i386 API-2500.bundle/Contents/MacOS/API-2500 | grep -i
checkcansubcomponent
```

This gives us the symbol name __Z31CheckCanSubComponentRunInternalR14PluginInstance. So let's start patching them all now! We'll use the --quiet flag on symset so we can just check success.

Code:

```
cd /Applications/Waves/Waves\ Plug-Ins/
for i in *; do symset --file="$i" --arch=i386 --
symbol=__Z31CheckCanSubComponentRunInternalR14PluginInstance --retvalue=0 --
quiet; done
```

You'll get some errors with this command, but that's just because some of the files we're iterating over in the loop are directorys, or don't have the symbol, etc. But the key is you'll get a bunch of successes too!

Now, I get a crash log when just running AU Lab (not under gdb)...

Code:

```
Exception Type:  EXC_BAD_ACCESS (SIGBUS)
Exception Codes: KERN_PROTECTION_FAILURE at 0x0000000000000000
Crashed Thread:  0  Dispatch queue: com.apple.main-thread

Thread 0 Crashed:  Dispatch queue: com.apple.main-thread
0   ???                             0000000000 0 + 0
1   com.WavesAudio.CLA Bass.7.1.0    0x24d4704c
WCStaticChainer_ALG::InstanciateAPlugin(CPlugInTypeInfo*) + 116
2   com.WavesAudio.CLA Bass.7.1.0    0x24d47443
WCStaticChainer_ALG::LoadPlugins() + 105
3   com.WavesAudio.CLA Bass.7.1.0    0x24d4753a
WCStaticChainer_ALG::UpdateAlgSelf() + 178
4   com.WavesAudio.CLA Bass.7.1.0    0x24d3df9f
```

```
WCCLA_Bass_ALG::UpdateAlgSelf() + 19
5   com.WavesAudio.WavesLib.7.1.1    0x251d91c4 cALGORITHM::UpdateAlg() + 40
6   com.WavesAudio.WavesLib.7.1.1    0x251dacbf
cALGORITHM::UpdateDSPCoefs(bool) + 27
7   com.WavesAudio.WavesLib.7.1.1    0x251db0cd
cALGORITHM::SetProcessActive(long) + 59
8   com.WavesAudio.CLA Bass.7.1.0    0x24d521e6 WP_SetProcessActive + 30
9   ...vesAudio.WaveShell-AU.7.1.1   0x24f16e38
CWSAU::StartAUShell(WavesComponentInfo const&) + 204
10  ...vesAudio.WaveShell-AU.7.1.1   0x24f1746c CWSAU::WavesInitialize() +
470
11  ...vesAudio.WaveShell-AU.7.1.1   0x24f1766f CWSAU::PostConstructor() + 95
12  ...vesAudio.WaveShell-AU.7.1.1   0x24f1b54d
ComponentEntryPoint<CWSAU>::Dispatch(ComponentParameters*, CWSAU*) + 59
13  ...ple.CoreServices.CarbonCore   0x93542ce5 CallComponentDispatch + 29
14  ...ple.CoreServices.CarbonCore   0x93542cc6 CallComponentOpen + 43
15  ...ple.CoreServices.CarbonCore   0x93540f94 OpenAComponent + 433
16  ...apple.audio.units.AudioUnit   0x90b07494 AudioComponentInstanceNew +
51
```

Now, a crash log is essentially a backtrace. It's the exact same as if you run the program under a debugger, let it break on the crash, and issued a backtrace. But here on the left, it also tells you in what binary the function call on the right corresponds to. So we can see it go up the chain, starting with AudioUnit, into CarbonCore, then the WaveShell-AU and the functions we remember from it (PostConstructor, WavesInitialize, and StartAUShell)... right above that, in SetProcessActive, it's in the CLA Bass plugin. But the very next call from SetProcessActive is in the WavesLib. Ahhh, so this plugin calls one of those external frameworks before the framework calls back into the plugin a few functions deeper. OK cool, so we've got a lot of information so far...

Now we'll look in particular at the most recent calls:

Code:

```
Thread 0 Crashed:  Dispatch queue: com.apple.main-thread
0   ???                              0000000000 0 + 0
1   com.WavesAudio.CLA Bass.7.1.0    0x24d4704c
WCStaticChainer_ALG::InstanciateAPlugin(CPlugInTypeInfo*) + 116
2   com.WavesAudio.CLA Bass.7.1.0    0x24d47443
```

```
WCStaticChainer_ALG::LoadPlugins() + 105
3   com.WavesAudio.CLA Bass.7.1.0    0x24d4753a
WCStaticChainer_ALG::UpdateAlgSelf() + 178
4   com.WavesAudio.CLA Bass.7.1.0    0x24d3df9f
WCCLA_Bass_ALG::UpdateAlgSelf() + 19
```

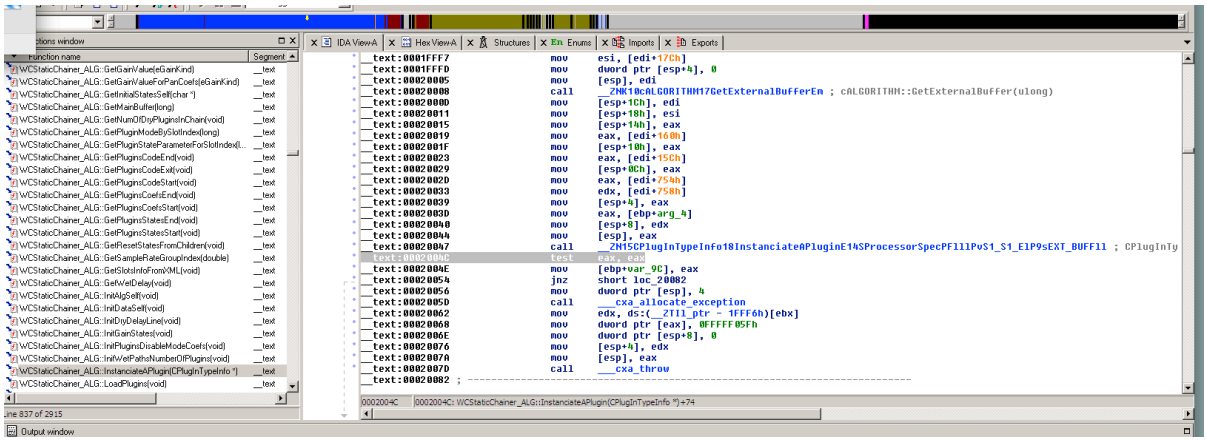
From this, we can tell that the function WCStaticChainer_ALG::InstanciateAPlugin(CPlugInTypeInfo*) in the CLA Bass plugin is where the crash is occurring. The fact that very top of the stack is showing as address 0x0 is very important.
0 ??? 0000000000 0 + 0
This means only a couple of things. The backtrace shows the value of eip, the program counter. So here, it says when it crashed, execution was at address 0x0... but you cannot execute __PAGEZERO. This means that a call or jump was made to a NULL address. It could also mean stack corruption. We can see this a little clearer from gdb too:

Code:

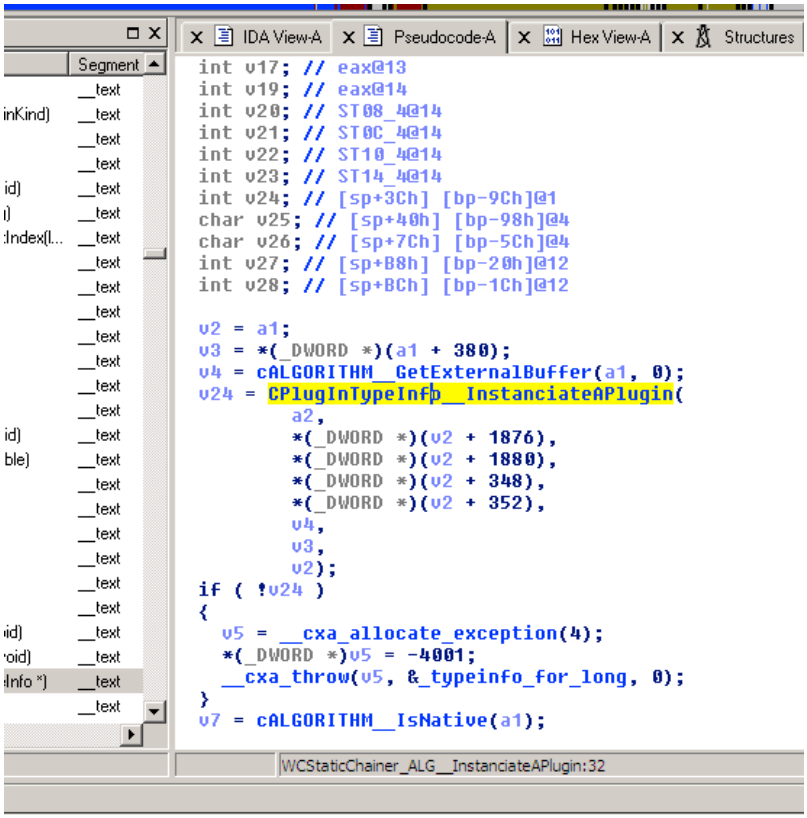
```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x00000000
0x00000000 in ?? ()
--------------------------------------------------------------------------------
-------------[regs]
  EAX: 008BE4B8  EBX: 254A0FF6  ECX: 00000000  EDX: 008BE004  o d I t s z a p
c
  ESI: 008BE000  EDI: 008DAE00  EBP: BFFFDB28  ESP: BFFFD9EC  EIP: 00000000
  CS: 001B  DS: 0023  ES: 0023  FS: 0000  GS: 000F  SS: 0023Error while
running hook_stop:
Cannot access memory at address 0x0
```

Here we can see the value of EIP is 0. OK, so time to investigate the function WCStaticChainer_ALG::InstanciateAPlugin(CPlugInTypeInfo*) in the CLA Bass plugin... load it up in IDA Pro and find that function from the list on the left, and then view it in regular IDA mode so we can see the offsets.

In the crash log, it says we're looking at address 0x24d4704c (to the left of the function name in the backtrace). At the very bottom of the crash log, you'll find image bases. You can grab the image base for CLA Bass and then subtract that from this address to get the offset 0x2004C. So look up this offset in IDA...



OK, if you remember, the way backtraces work is by showing the first address (in this case 0x0) as the current EIP value, and all the other address are the return addresses for function calls on the stack. I mentioned in another tutorial the way a call instruction works... it pushes the ***next*** instruction onto the stack and then jumps to the address that is being called. It's these pushed address that are being shown in the backtrace. So the offset it brought us to is actually the address after the call it entered, in this case __ZN15CPlugInTypeInfo18InstanciateAPluginE14SProcessorSpecPFlllPvS1_S1_ElP9sEXT_BUFFll, or actually CPlugInTypeInfo__InstanciateAPlugin if you enter decompilation mode now 😀



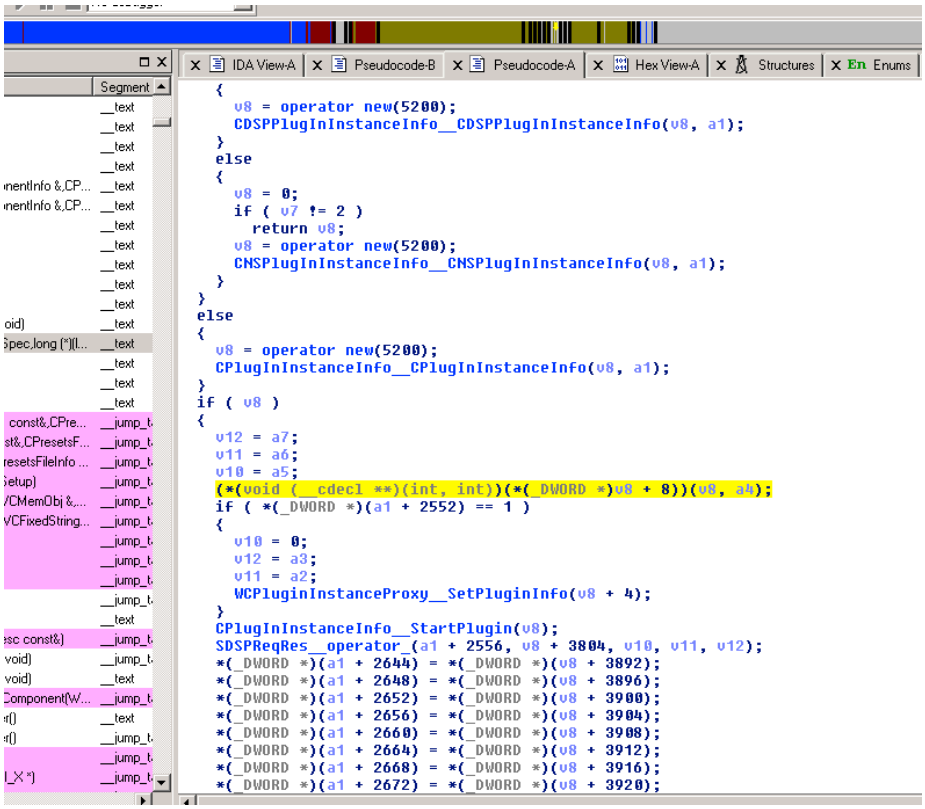
So basically, what this means is that somewhere inside the CPlugInTypeInfo__InstanciateAPlugin function, we are jumping to a NULL pointer, or the stack is getting severely corrupted... most likely the former. So we have to scan in this function to see what might be going wrong.

Nothing really jumps out at you when you look at the code, at least nothing obvious. So probably a good idea is to break on this function in gdb, since it seems that the backtrace we got from the crash is not very complete...
So if you load gdb with AU Lab as the target, before running the debugger, set a breakpoint on the last first function from the backtrace:
gdb$ hb WCStaticChainer_ALG::InstanciateAPlugin(CPlugInTypeInfo*)
and then step through until you get to the CPlugInTypeInfo::InstanciateAPlugin function and then step into it.
Now this is the function that's causing problems, but there are a bunch of function calls in this function, and one of them is causing a problem. I usually use the process of elimination here... Step through the whole function, not following any calls, just stepping over them, until you find a call that causes a crash. Then restart gdb, step into that call, and do the same thing, until you've narrowed down the function.

Next step is to find that function in IDA Pro so we can analyze it further.

Best way to do this is to follow along in IDA Pro as you're stepping through in gdb. Now, this next part is a little complicated, but is a skill that I think is very important for a reverser to have. It's being able to look at a function argument, and tracing it back to it's source, where it's value was assigned. I would encourage you, if you are not familiar with this process, to do it a few times to make sure you understand how it's done.

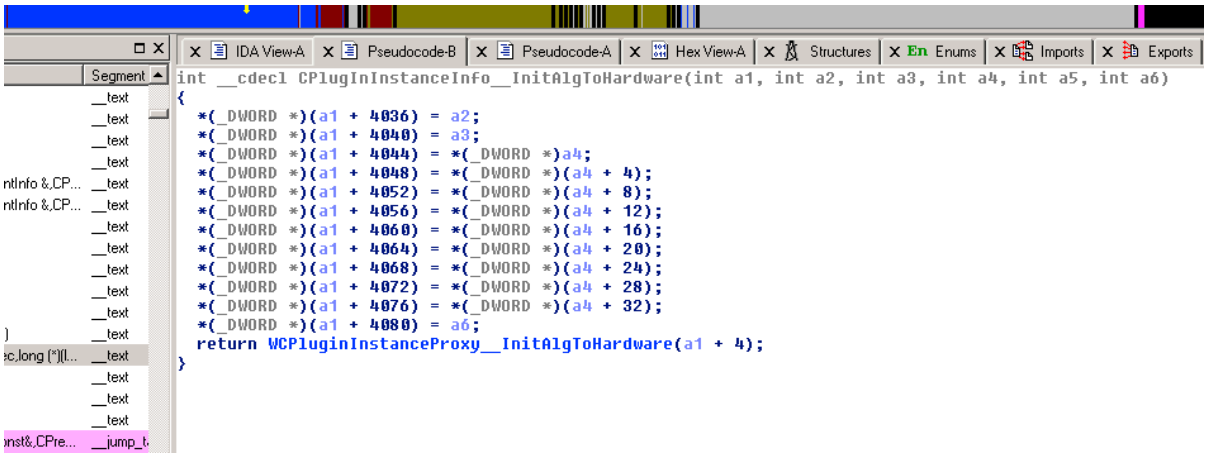
OK, so starting at the CPlugInTypeInfo__InstanciateAPlugin function, we'll step through in gdb and find the crashes, and then step into them in IDA Pro. So here's the first function that it crashes on:



You may remember from the other tutorial (DUY) how to follow this function here?
v8 = operator new(5200);
CPlugInInstanceInfo__CPlugInInstanceInfo(v8, a1);

HINT: This is where v8 is instantiated, follow that function CPlugInInstanceInfo and check the vtable value that gets assigned, then follow the address that's 8 bytes into the list. OR, look again at the DUY tut for a refresher 😉

Leads us to this function here:


IDA View-A
Pseudocode-B
Pseudocode-A
Hex View-A
Structures
Enums
Imports
Exports
Segment
int __cdecl CPlugInInstanceInfo__InitAlgToHardware(int a1, int a2, int a3, int a4, int a5, int a6)
{
*(_DWORD *)(a1 + 4036) = a2;
*(_DWORD *)(a1 + 4040) = a3;
*(_DWORD *)(a1 + 4044) = *(_DWORD *)a4;
*(_DWORD *)(a1 + 4048) = *(_DWORD *)(a4 + 4);
*(_DWORD *)(a1 + 4052) = *(_DWORD *)(a4 + 8);
*(_DWORD *)(a1 + 4056) = *(_DWORD *)(a4 + 12);
*(_DWORD *)(a1 + 4060) = *(_DWORD *)(a4 + 16);
*(_DWORD *)(a1 + 4064) = *(_DWORD *)(a4 + 20);
*(_DWORD *)(a1 + 4068) = *(_DWORD *)(a4 + 24);
*(_DWORD *)(a1 + 4072) = *(_DWORD *)(a4 + 28);
*(_DWORD *)(a1 + 4076) = *(_DWORD *)(a4 + 32);
*(_DWORD *)(a1 + 4080) = a6;
return WCPluginInstanceProxy__InitAlgToHardware(a1 + 4);
}

And as we follow this up to the next line here:

return WCPluginInstanceProxy__InitAlgToHardware(a1 + 4);

and follow that in we get to here, which is where the crash happens:

OK, so this is not so easy to read... But from gdb, we see that this function looks like

Code:

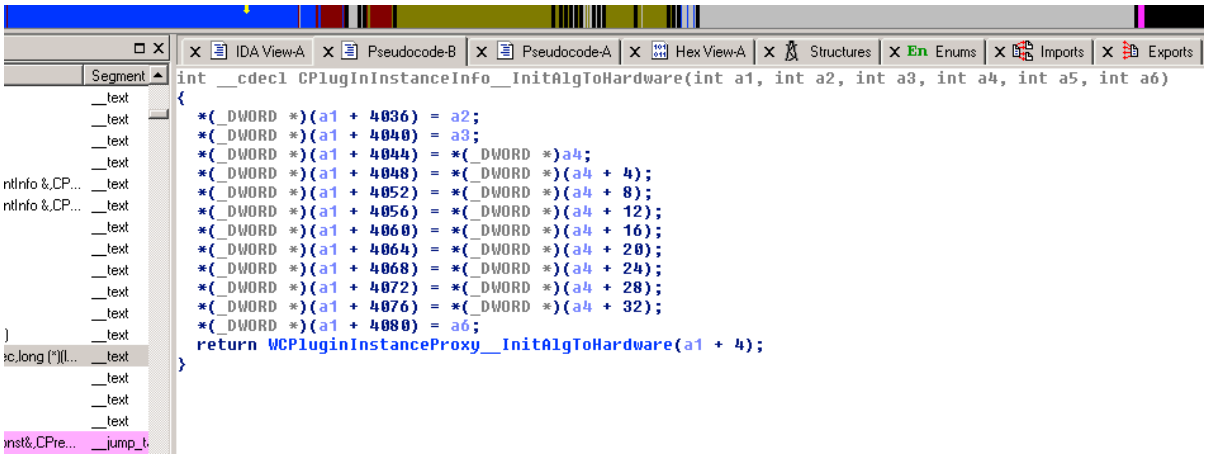
```
0x370d39c
<_ZNK21WCPluginInstanceProxy17InitAlgToHardwareEPFlllPvS0_S0_Ell>:   push   e
bp
0x370d39d
<_ZNK21WCPluginInstanceProxy17InitAlgToHardwareEPFlllPvS0_S0_Ell+1>:   mov
 ebp,esp
0x370d39f
<_ZNK21WCPluginInstanceProxy17InitAlgToHardwareEPFlllPvS0_S0_Ell+3>:   mov
 edx,DWORD PTR [ebp+0x8]
0x370d3a2
<_ZNK21WCPluginInstanceProxy17InitAlgToHardwareEPFlllPvS0_S0_Ell+6>:   mov
 ecx,DWORD PTR [edx+0x4a8]
0x370d3a8
<_ZNK21WCPluginInstanceProxy17InitAlgToHardwareEPFlllPvS0_S0_Ell+12>:   mov
  eax,DWORD PTR [edx+0x4ac]
```

```
0x370d3ae
<_ZNK21WCPluginInstanceProxy17InitAlgToHardwareEPFlllPvS0_S0_Ell+18>:    mov
  DWORD PTR [ebp+0x8],eax
0x370d3b1
<_ZNK21WCPluginInstanceProxy17InitAlgToHardwareEPFlllPvS0_S0_Ell+21>:    mov
  ecx,DWORD PTR [ecx+0x98]
0x370d3b7
<_ZNK21WCPluginInstanceProxy17InitAlgToHardwareEPFlllPvS0_S0_Ell+27>:    leave

0x370d3b8
<_ZNK21WCPluginInstanceProxy17InitAlgToHardwareEPFlllPvS0_S0_Ell+28>:    jmp
  ecx
```

and this "jmp ecx" is where the jump to NULL happens, because ecx is 0. As we step through this function in gdb, we can see that where ecx gets assigned a 0 is at the "mov ecx, [ecx+0x98]". In IDA, this would be the "+ 152". OK, so this is where we need to follow the "+ 152" back to it's source.

So looking at the line, it says (*(_DWORD *)(a1 + 1192) + 152) is the function being called. So in other words, it reads: jump to the pointer at offset 152 of the pointer at offset 1192 of argument 1. This is where it gets a bit tough to explain, but think of it as a math function, or if you're familiar with programming, as a complex assignment. Basically, it's taking argument 1 and getting the value at 1192. THEN, it's taking that result, and calling the pointer at 152. So we need to trace what a1 is, and the value that gets assigned to 1192. So back out into the last function here so we can see what argument 1 is:


IDA View-A
Pseudocode-B
Pseudocode-A
Hex View-A
Structures
Enums
Imports
Exports
Segment
int __cdecl CPlugInInstanceInfo__InitAlgToHardware(int a1, int a2, int a3, int a4, int a5, int a6)
{
*(_DWORD *)(a1 + 4036) = a2;
*(_DWORD *)(a1 + 4040) = a3;
*(_DWORD *)(a1 + 4044) = *(_DWORD *)a4;
*(_DWORD *)(a1 + 4048) = *(_DWORD *)(a4 + 4);
*(_DWORD *)(a1 + 4052) = *(_DWORD *)(a4 + 8);
*(_DWORD *)(a1 + 4056) = *(_DWORD *)(a4 + 12);
*(_DWORD *)(a1 + 4060) = *(_DWORD *)(a4 + 16);
*(_DWORD *)(a1 + 4064) = *(_DWORD *)(a4 + 20);
*(_DWORD *)(a1 + 4068) = *(_DWORD *)(a4 + 24);
*(_DWORD *)(a1 + 4072) = *(_DWORD *)(a4 + 28);
*(_DWORD *)(a1 + 4076) = *(_DWORD *)(a4 + 32);
*(_DWORD *)(a1 + 4080) = a6;
return WCPluginInstanceProxy__InitAlgToHardware(a1 + 4);
}

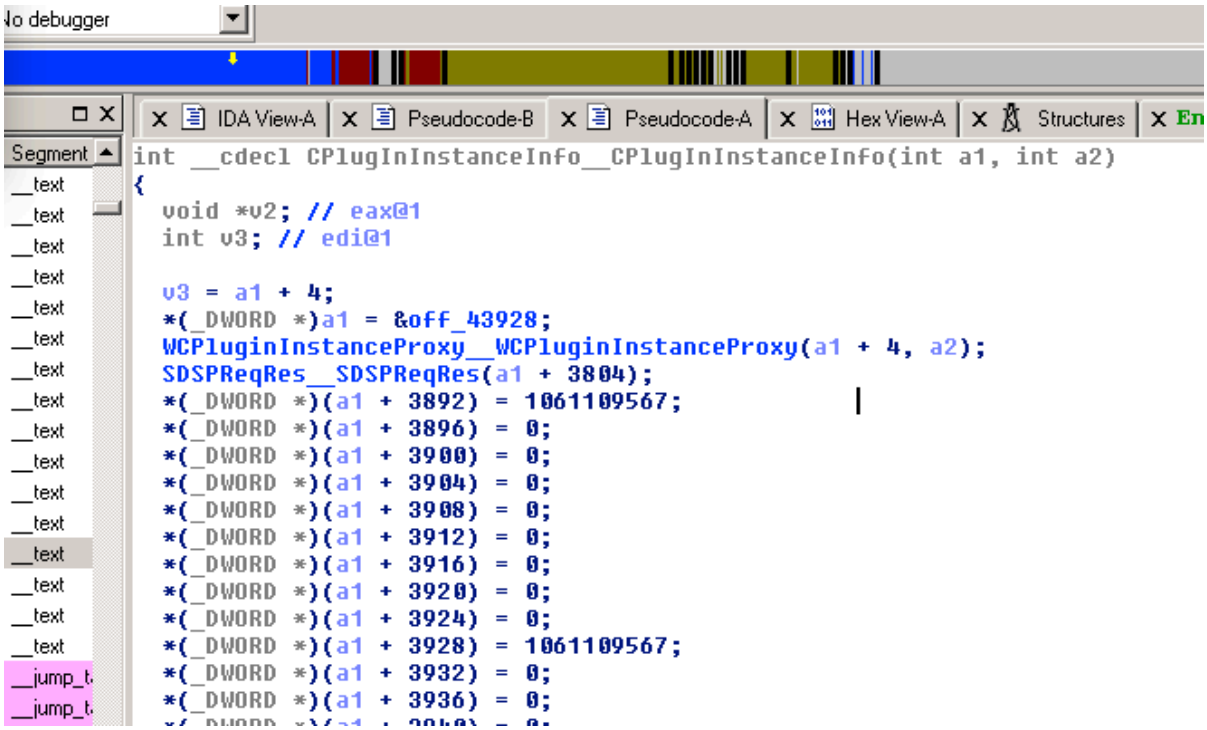
OK, so here, the argument that gets passed in to become argument 1 is a1 + 4. So we need to find out what a1 is, then when we have a1, get a1 + 4, because that is what we have to get the value at 1192 from, and then finally value 152 from that. Take notes on this if you have to, but kudos if you can keep it straight in your head

So we back out to the previous function again, and see the argument 1 passed in is v8:

(*(void (__cdecl **)(int, int))(*(_DWORD *)v8 + 8))(v8, a4);

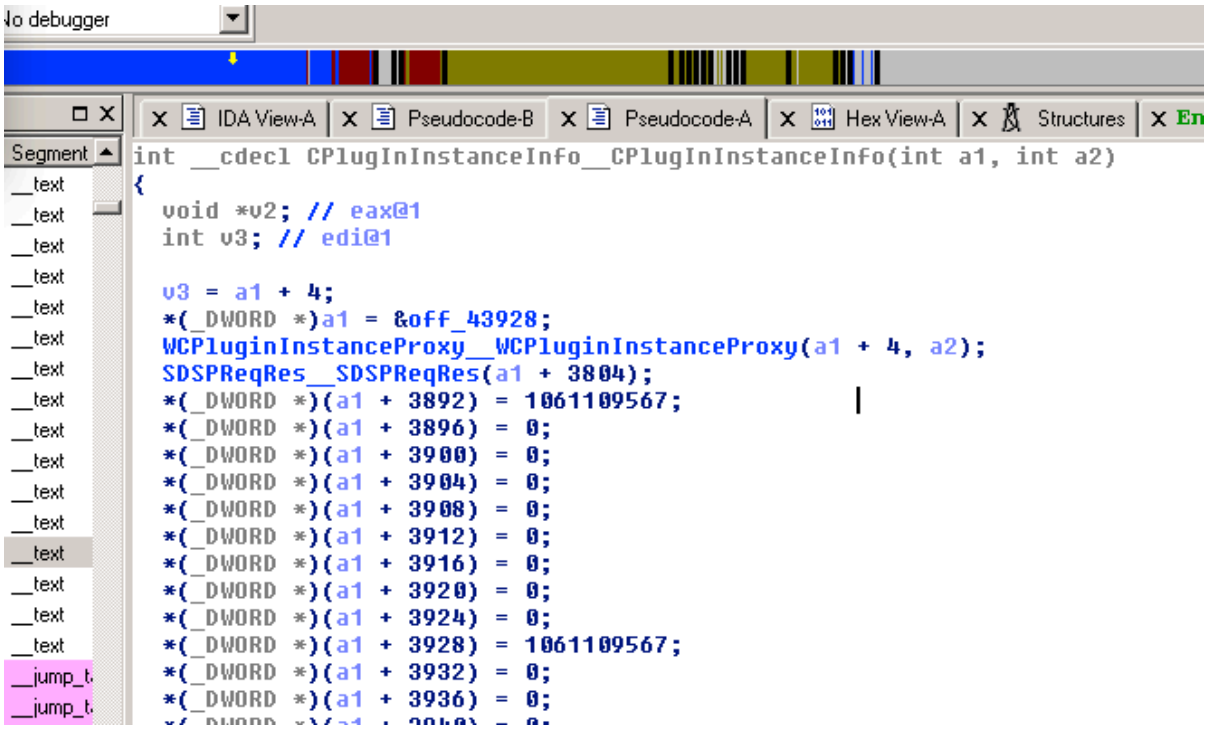
OK, so here we know that (((v8 + 4) + 1192) + 152) is what is zero, and eventually causes the crash. So now we need to follow v8, and see where the NULL value causing the crash is coming from. So step into the function CPlugInInstanceInfo here:
v8 = operator new(5200);
CPlugInInstanceInfo__CPlugInInstanceInfo(v8, a1);
because this is where v8 is created with the "new" function, and then initialized.

It brings us here:

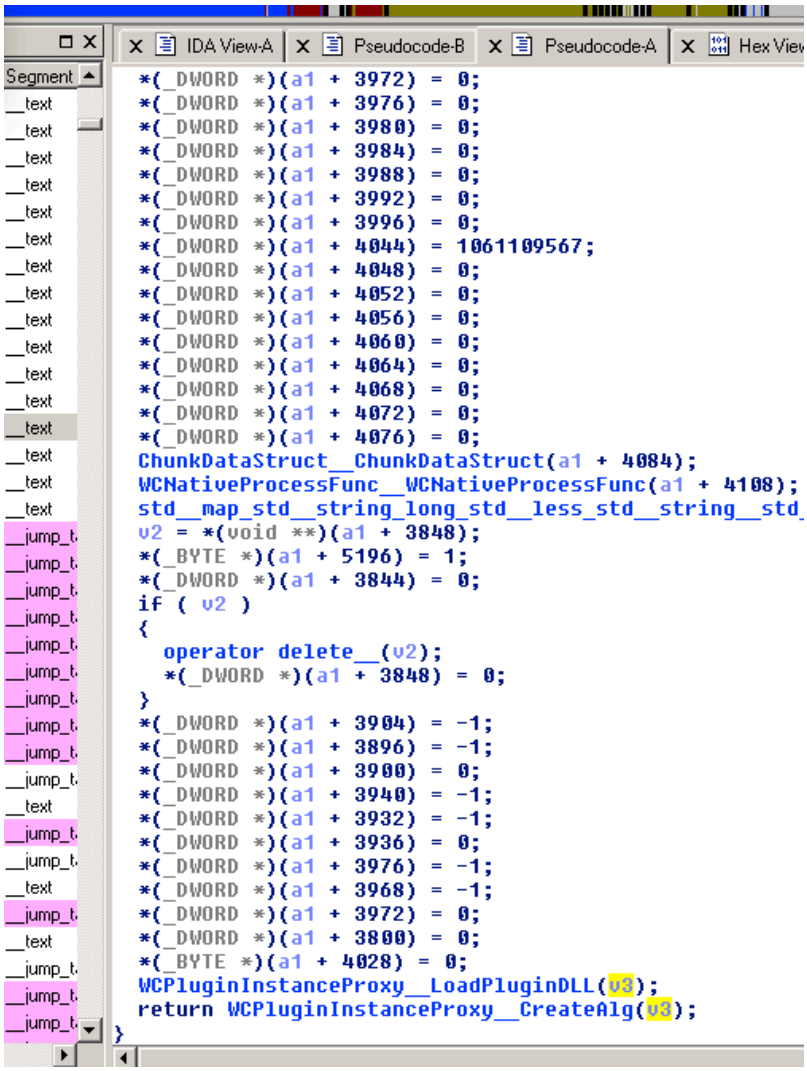

No debugger
IDA View-A
Pseudocode-B
Pseudocode-A
Hex View-A
Structures
Segment
int __cdecl CPlugInInstanceInfo__CPlugInInstanceInfo(int a1, int a2)
{
void *v2; // eax@1
int v3; // edi@1
v3 = a1 + 4;
*(_DWORD *)a1 = &off_43928;
WCPluginInstanceProxy__WCPluginInstanceProxy(a1 + 4, a2);
SDSPReqRes__SDSPReqRes(a1 + 3804);
*(_DWORD *)(a1 + 3892) = 1061109567;
*(_DWORD *)(a1 + 3896) = 0;
*(_DWORD *)(a1 + 3900) = 0;
*(_DWORD *)(a1 + 3904) = 0;
*(_DWORD *)(a1 + 3908) = 0;
*(_DWORD *)(a1 + 3912) = 0;
*(_DWORD *)(a1 + 3916) = 0;
*(_DWORD *)(a1 + 3920) = 0;
*(_DWORD *)(a1 + 3924) = 0;
*(_DWORD *)(a1 + 3928) = 1061109567;
*(_DWORD *)(a1 + 3932) = 0;
*(_DWORD *)(a1 + 3936) = 0;

So at this point, just to keep it straight, we know it's (((a1 +4) + 1192) + 152) that is the NULL pointer. So we see the line:
WCPluginInstanceProxy__WCPluginInstanceProxy(a1 + 4, a2);
and that's passing in the first argument as a1+4, so it could be then assigning some value to the 1192 and eventually the 152 in there... Go ahead and poke around in there, and see if you can follow the concepts I'm presenting here.

Turns out that function was doing nothing with the 1192 or the 152 values... So back in the same function here:


No debugger
IDA View-A
Pseudocode-B
Pseudocode-A
Hex View-A
Structures
Segment
__text
__jump_t
int __cdecl CPlugInInstanceInfo__CPlugInInstanceInfo(int a1, int a2)
{
void *v2; // eax@1
int v3; // edi@1
v3 = a1 + 4;
*(_DWORD *)a1 = &off_43928;
WCPluginInstanceProxy__WCPluginInstanceProxy(a1 + 4, a2);
SDSPReqRes__SDSPReqRes(a1 + 3804);
*(_DWORD *)(a1 + 3892) = 1061109567;
*(_DWORD *)(a1 + 3896) = 0;
*(_DWORD *)(a1 + 3900) = 0;
*(_DWORD *)(a1 + 3904) = 0;
*(_DWORD *)(a1 + 3908) = 0;
*(_DWORD *)(a1 + 3912) = 0;
*(_DWORD *)(a1 + 3916) = 0;
*(_DWORD *)(a1 + 3920) = 0;
*(_DWORD *)(a1 + 3924) = 0;
*(_DWORD *)(a1 + 3928) = 1061109567;
*(_DWORD *)(a1 + 3932) = 0;
*(_DWORD *)(a1 + 3936) = 0;

we can see that at the top, v3 is being assigned the address a1 + 4.... hmm so we need to follow v3, because conceptually we can adjust our formula to include v3 instead of a1 + 4, so it looks like this: ((v3 + 1192) + 152) being the NULL address. If you look all the way at the bottom of the function, you'll see some calls with v3 as the argument:


IDA View-A
Pseudocode-B
Pseudocode-A
Hex View
Segment
*(_DWORD *)(a1 + 3972) = 0;
*(_DWORD *)(a1 + 3976) = 0;
*(_DWORD *)(a1 + 3980) = 0;
*(_DWORD *)(a1 + 3984) = 0;
*(_DWORD *)(a1 + 3988) = 0;
*(_DWORD *)(a1 + 3992) = 0;
*(_DWORD *)(a1 + 3996) = 0;
*(_DWORD *)(a1 + 4044) = 1061109567;
*(_DWORD *)(a1 + 4048) = 0;
*(_DWORD *)(a1 + 4052) = 0;
*(_DWORD *)(a1 + 4056) = 0;
*(_DWORD *)(a1 + 4060) = 0;
*(_DWORD *)(a1 + 4064) = 0;
*(_DWORD *)(a1 + 4068) = 0;
*(_DWORD *)(a1 + 4072) = 0;
*(_DWORD *)(a1 + 4076) = 0;
ChunkDataStruct__ChunkDataStruct(a1 + 4084);
WCNativeProcessFunc__WCNativeProcessFunc(a1 + 4108);
std__map_std__string_long_std__less_std__string__std_
v2 = *(void **)(a1 + 3848);
*(_BYTE *)(a1 + 5196) = 1;
*(_DWORD *)(a1 + 3844) = 0;
if (v2)
{
operator delete__(v2);
*(_DWORD *)(a1 + 3848) = 0;
}
*(_DWORD *)(a1 + 3904) = -1;
*(_DWORD *)(a1 + 3896) = -1;
*(_DWORD *)(a1 + 3900) = 0;
*(_DWORD *)(a1 + 3940) = -1;
*(_DWORD *)(a1 + 3932) = -1;
*(_DWORD *)(a1 + 3936) = 0;
*(_DWORD *)(a1 + 3976) = -1;
*(_DWORD *)(a1 + 3968) = -1;
*(_DWORD *)(a1 + 3972) = 0;
*(_DWORD *)(a1 + 3800) = 0;
*(_BYTE *)(a1 + 4028) = 0;
WCPluginInstanceProxy__LoadPluginDLL(v3);
return WCPluginInstanceProxy__CreateAlg(v3);
}

Follow the first one in, WCPluginInstanceProxy__LoadPluginDLL, and now remember here that our formula has changed to ((a1 + 1192) + 152)... and in this function we see:

bingo! WCPluginLoader__LoadPlugin is getting passed the 1192 as an argument, so let's follow that one

in! Now remember to adjust your formula for this function: (a1 + 152) is what's crashing. But looking through here, it looks very similar to the WaveShell patching we did... It's got a function halfway down
WCPluginLoader__LoadAndCachePluginWaveslib
as well as later on:
wvPACEWrapper__PACEIsAuthorizedMachoShlibWrap
We haven't patched these functions in any of the plugins themselves, only the WaveShells! Right here we could go patch these just like we did in the waveshells and it would probably work. But for those interested in how to finish tracking down the crash, keep poking around here looking for (a1 + 152). You'll notice what happens is that if any one of those functions don't validate, an exception is thrown:

```
v9 = __cxa_allocate_exception(4);
*(_DWORD *)v9 = v4;
__cxa_throw(v9, &_typeinfo_for_long, 0);
```

And that exception getting thrown means that at the bottom of this function, the "WCPluginLoader__AssignPluginInterface" never gets called. And if you notice, the WCPluginLoader__AssignPluginInterface takes a1 as a parameter, and inside the function, you'll see where (a1 + 152) would get assigned the WP_InitAlg function pointer

```
WCFixedString_63__WCFixedString(&v36, "WP_GetPluginInfo");
*(_DWORD *)(a1 + 136) = *(_DWORD *)std__map_WCFixedString_63__void__std__less_WCFixedString_63__std__allocator_std__pair_WCFixedString_63__cons
                                   a1 + 1344,
                                   &v36);
WCFixedString_63__WCFixedString(&v37, "WP_ProcessFunctionLock");
*(_DWORD *)(a1 + 140) = *(_DWORD *)std__map_WCFixedString_63__void__std__less_WCFixedString_63__std__allocator_std__pair_WCFixedString_63__cons
                                   a1 + 1344,
                                   &v37);
WCFixedString_63__WCFixedString(&v38, "WP_GetViewRect");
*(_DWORD *)(a1 + 144) = *(_DWORD *)std__map_WCFixedString_63__void__std__less_WCFixedString_63__std__allocator_std__pair_WCFixedString_63__cons
                                   a1 + 1344,
                                   &v38);
WCFixedString_63__WCFixedString(&v39, "WP_InitAlg");
*(_DWORD *)(a1 + 148) = *(_DWORD *)std__map_WCFixedString_63__void__std__less_WCFixedString_63__std__allocator_std__pair_WCFixedString_63__cons
                                   a1 + 1344,
                                   &v39);
WCFixedString_63__WCFixedString(&v40, "WP_InitAlgToHardware");
*(_DWORD *)(a1 + 152) = *(_DWORD *)std__map_WCFixedString_63__void__std__less_WCFixedString_63__std__allocator_std__pair_WCFixedString_63__cons
                                   a1 + 1344,
                                   &v40);
WCFixedString_63__WCFixedString(&v41, "WP_InitGUI");
*(_DWORD *)(a1 + 156) = *(_DWORD *)std__map_WCFixedString_63__void__std__less_WCFixedString_63__std__allocator_std__pair_WCFixedString_63__cons
                                   a1 + 1344,
                                   &v41);
WCFixedString_63__WCFixedString(&v42, "WP_InitProcessing");
*(_DWORD *)(a1 + 164) = *(_DWORD *)std__map_WCFixedString_63__void__std__less_WCFixedString_63__std__allocator_std__pair_WCFixedString_63__cons
                                   a1 + 1344,
                                   &v42);
WCFixedString_63__WCFixedString(&v43, "WP_LoadFromBuffer");
*(_DWORD *)(a1 + 168) = *(_DWORD *)std__map_WCFixedString_63__void__std__less_WCFixedString_63__std__allocator_std__pair_WCFixedString_63__cons
                                   a1 + 1344,
                                   &v43);
```

So when an exceptions is thrown, a1 + 152 never gets initialized so it remains 0, and later on the program still tries to call it and jumps to address 0x0, getting the crash we see in the crash log. Tracking down these kind of crashes is a lot of fun once you learn how, and I really hope that after this tutorial, you guys will see the value of a crash log!

Whew! Hope you guys were able to follow this post, and hope you had as much fun as I did! So basically, patching two additional functions on the plugins that crash is the next step. I use symset, and I hope you all remember how to find the symbol to use...

You'll find the isWavesLibAuthorized symbol in the actual plugin, but you'll notice if you try to use symset or search for the PACEIsAuthorized function yourself, it's not located in the plugin... it's actually in the ArtistLib framework.

At this point, we've fixed the IsWavesLibAuthorized to always return 0, probably the best way to double-check that this function is patched in all the files is to run symset like this:

Code:

```
cd /Applications/Waves/Waves\ Plug-Ins
for i in *; do symset --file="$i" --arch=i386 --
symbol=__Z20IsWavesLibAuthorizedRN4wvFM8WCStPathE --retvalue=0 --quiet; done
```

This will search for the symbol in ALL plugins and patch it if it finds it. Probably the safest bet. You could go ahead and do this for the PACEIsAuthorizedMachOSLibWrap function too. But even after all this, we can see it's still crashing, and crashing when loading the frameworks...

So look at the frameworks in the Finder by Showing Package Contents... Ahhh notice that there's a WavesLib in the base framework folder, but also one in framework/Version/A/WavesLib, and they have two different file sizes. We know that unpace strips ppc code as well as symbols and segments from the file, so the unpaced one is the smaller one. Let's quickly check to see what is still PACE protected in the plugins folder...

Code:

```
cd /Applications/Waves/Waves\ Plug-Ins
grep -i -r sgemexam .
```

Run that command and you'll see the frameworks are still protected... So obviously when we passed the frameworks as parameters to unpace, it only unpaced the one in the root of the folder, not the one in /Versions/A/WavesLib.... If you glance at other frameworks (not waves frameworks, normal frameworks from other apps), you'll see that the one at the base is usually a *symlink* that points into Versions/A/. This one is not, so unpace unpacked the file, but the other one is the one actually getting loaded.

Two ways to fix this... delete the one in Versions/A/ and copy the one from the base directory there instead. That way they're both the unpaced versions. Or move the unpaced one into Verions/A/, deleting the one that was already there, and then creating a symlink to the base folder. Either way is up to you. Do this for all the frameworks in the plugins directory. For Multirack, patch the three symbols in the dylds by using the nm command to find their offsets. For example:

Code:

```
nm /path/to/MultiRack.app/Contents/MacOS/MultiRackCore.dylib | grep -i
paceisauth
```

which returns:

Code:

```
000ff6cc t _PACEIsAuthorized
000fe1c6 t _PACEIsAuthorizedMachoShlib
0010000e t _PACEIsValidSerialNum
```

```
0006cbd8 T __ZN13wvPACEWrapper20PACEIsAuthorizedWrapERmhhhPKhPmS3_Ps
0006cc22 T __ZN13wvPACEWrapper30PACEIsAuthorizedMachoShlibWrapEP10__CFBundle
```

The last two symbols are what we need to patch (along with iswaveslib...) and their offsets are 0x0006cbd8 and 0x0006cc22. To patch them, open up hex fiend and use the "jump to offset" command. Then put hex fiend in overwrite mode and overwrite the first line at each offset starting at 5589 with 31c0c3. Patching these 3 in the dyld, iswavelibauth in the bundle, and both paceisauth calls in framework did the trick.

Congratulations, you have stuck with it and now all the Waves plug-ins have been cracked.

Addendum:

It is 100% possible to unpace DLA.bundle and PS22.bundle ***without*** having access to a TDM rig. Additionally, I have found that it's possible to batch unpace the entire Waves folder without needing to use AudioUnitHosting.app OR selecting each and every plugin from a list, and instead just letting unpace do it's own thing.

So the reason it is hanging on "Loading Library" when attempting to unpace is because the plugins load the frameworks, and the frameworks are running initialization code that causes unpace to hang. So we need to make some modifications to the frameworks so that they do not run any code upon loading.

First things first, make a backup of the frameworks and move them to a safe location (folder on the desktop, etc.). Then, we need to zero out the number of initializer methods in the framework, so that dyld does not run any code in the framework before unpace finishes doing it thing. So let's run otool on the WavesLib 7.1.framework to examine the load commands, and let's look for the section __mod_init_funcs. It's this section that contains the loading code for apps, bundles. frameworks, everything.

Code:

```
Section
  sectname __mod_init_func
   segname __DATA
      addr 0x0023aa34
      size 0x00000314
    offset 2337332
     align 2^2 (4)
    reloff 0
    nreloc 0
     flags 0x00000009
reserved1 0
reserved2 0
```

So with that, we can see the structure of the section header. Now open up the WavesLib binary in 0xED, and search for the __mod_init_funcs string. It may be easier to lipo the binary to i386 first:

Code:

```
lipo -thin i386 /Path/To/WavesLib\ 7.1.framework/Versions/A/WavesLib -output
/Path/To/WavesLib\ 7.1.framework/Versions/A/WavesLib
```

OK, once you've done that, open it in 0xED. Now do a search for the sectname ("mod_init_funcs"). You'll see in 0xED that it's a string of characters that ends with a 0. After the 0, you'll see the __DATA string, which corresponds to the segname in the otool output. Next in the otool output you see address of 0x23AA34 and then size of 0x314... Looking in the 0xED window, you can easily see the bytes that correspond to the size, at offset 0x358. So we need to zero that out. Change the bytes 14 03 to be 00 00.

Perfect, now there will be no code ran in the frameworks before unpace because when dyld loads this executable, it'll show a __mod_init_func size of 0! Do this for each one of the frameworks, and these will be our dummy frameworks (for DLA.bundle and PS22.bundle, the only framework that needs to be modified is the WavesLib 7.1.framework). Using these frameworks (by placing them in the same directory as the Waves .bundle files) you can successfully unpace the plugins without using any custom host! So just:

Code:

```
for i in *.bundle; do unpace -f "$i"; done;
```

Once you have unpaced the plugins, trash the dummy frameworks and return the originals to their places... Enjoy!

If you want to completely do waves with the GTR and GTRSolo, you will have to manually patch the addresses for

Code:

```
IsWavesLibAuthorized(wvFM::WCStPath&)
wvPACEWrapper::PACECloseProtectionWrap(unsigned long&)
wvPACEWrapper::PACEIsAuthorizedMachoShlibWrap(__CFBundle*)
```

as 31c0c3. But you will have to subtract 0x1000 fom the base address which you get from disassembling the file in OTX.

Although this is for version 7.1, this same applies for version 8 also, as long as they don’t apply the new Eden protection system.

Concepts for Writing an Emulator

By unsanity MSJ

This is a bit more of an advanced tutorial, but I believe that very soon this will be the only way to crack some of the more advanced protection out there. Sometimes, simple hex cracks are not enough, as the protection is more complex. I came across this while looking at Pod Farm last week, and while I feel that a tutorial on writing an emulator for Pod Farm would be way to complex and difficult to teach, I believe I may have found a good starting point.

The program we will be looking at is DXO Optics Pro 6.2. Now, for this particular program, it has already been cracked, and it is not even necessary to write an emulator to crack it. That's not the point. The point of this tut is to teach you the basic concepts of writing an emulator to bypass protection, as well as determining whether or not it's feasible to do so.

Target:
DXO Optics Pro 6.2 (can be downloaded from http://www.dxo.com)

Tools Required:
XCode (and familiarity with it)
IDA Pro or **OTX**
GDB
class-dump, found here: http://www.codethecode.com/projects/class-dump/

Required Knowledge:
Some programming experience with **ObjC** or **C**

OK. So with this target, we won't ***really*** be writing an emulator, in the strictest sense of the word. But writing an emulator for, say, a dongle protection would be accomplished using the very same techniques I will be presenting here.

Since this is an advanced tutorial (and for the sake of brevity), I will skip over the basic parts like how to find where the protection is. For this target, the protection is in DXActivation.framework, and this is the framework we will be emulating. This is a cocoa framework, and as such, we can get the header files by using the class-dump utility on it. We'll go ahead and see what it does by dumping the headers to a folder on our desktop.

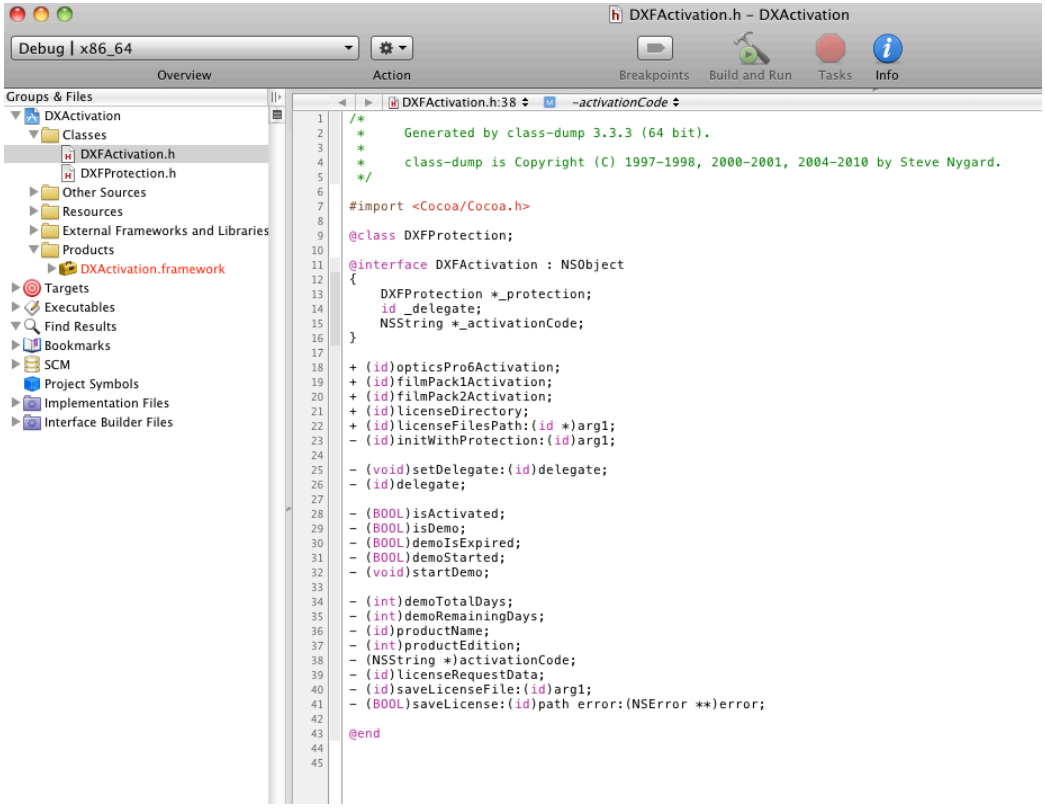
Code:

```
class-dump -H -o ~/Desktop/DXFActivation_Headers
/Applications/DXOOpticsPro6.app/Contents/Frameworks/DXFActivation.framework/V
ersions/A/DXFActivation
```

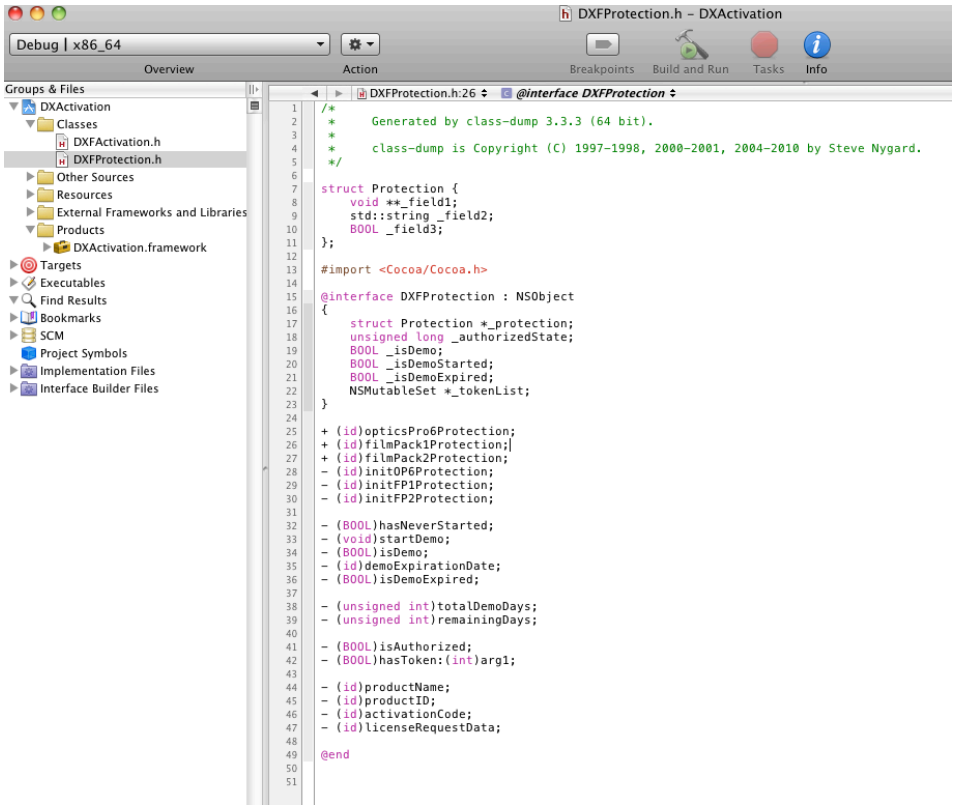
Now, after looking through the dumped headers, we can clearly see there are 2 classes in the framework, DXFActivation and DXFProtection. So to begin, we'll start by copying the header files for DXFActivation and DXFProtection into a new XCode project.

Open XCode, and start a new project from the Cocoa Framework template. Call the project "DXFActivation", so that it creates a framework of the same name when compled (since we'll be replacing the current DXFActivaion.framework). Once it's open, just drag and drop the DXFActivation.h and DXFProtection.h into the Classes folder (be sure to select the checkbox up top of the sheet that comes up to "copy" them into the project). Now, you can go through and clean up the header files from things that we won't be using... which will be anything GUI related. We can add in our own implementation later if we need to.

Here's DXFActivation after cleanup:



And here's DXFProtection:

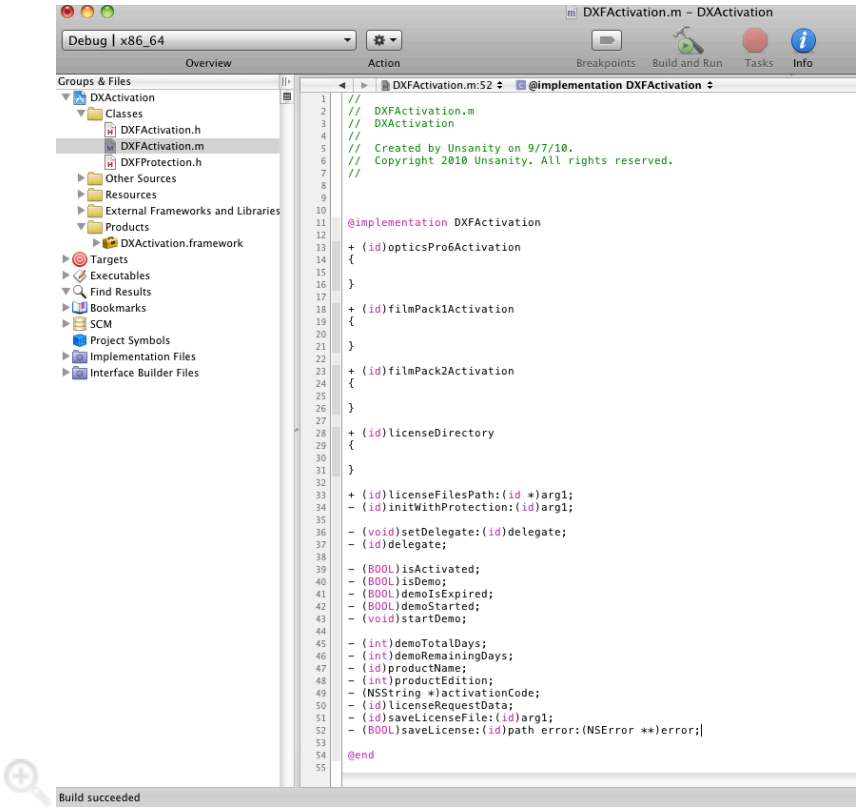


You'll notice that one of the ivars in DXFProtection is a "struct Protection *_protection". That is taken from the CDStructures.h file that class-dump dumped. We went ahead and copied the declaration from that file and just pasted it into the top of DXFProtection.h. We don't know it's necessary yet, but just to keep everything in these two files for now, for simplicity's sake.

I changed the "basic_string" in struct Protection to a std::string, this is because it's the only way to compile it and it'll still work. I'm sure that later in this tutorial we may even change it to an NSString 😉. When writing an emulator, you don't have to implement the protection the same way as in the current program... if they use std::map's, you can use an NSDictionary. You can re-implement it any way you want. This is why they are so much fun to write!

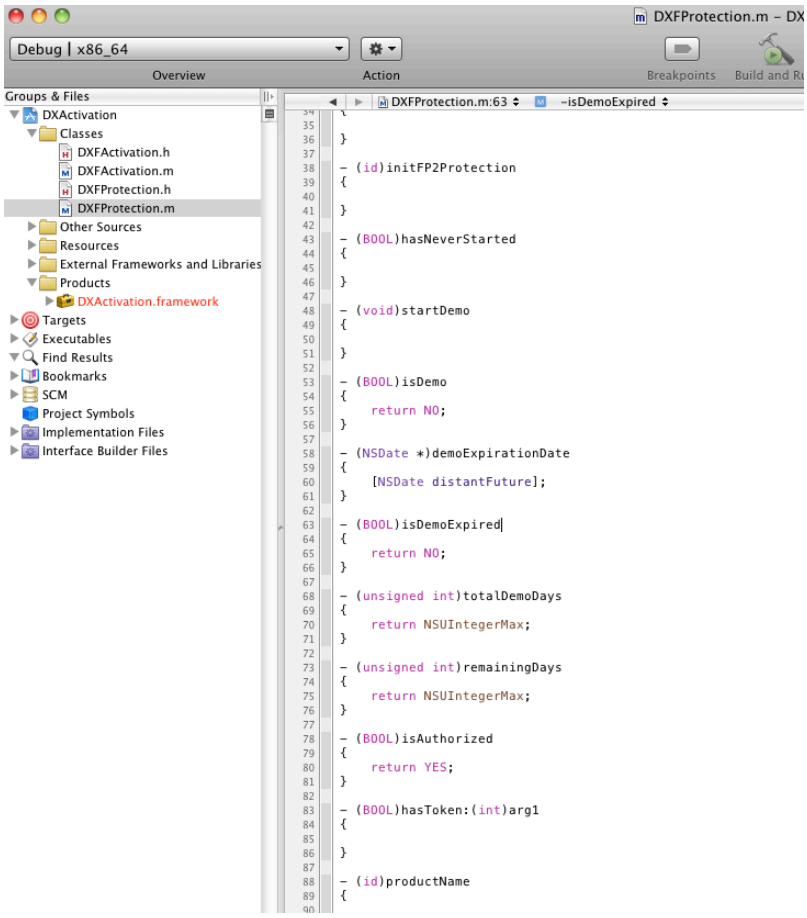
So class-dump got the headers for us. Now we'll be using OTX to get the implementation details. First, we need to write functions for the methods we have in the headers. So create a new .m file called DXFActivation.m (by right-clicking on the Classes folder, and selecting "Add->New File"). Choose "Objective-C Class" from the options and click Next. Then, uncheck the option that says to also create e a .h file as well, because we already have a .h file. Name it DXFActivation.m.

Here, I just copy-paste the functions from the header into the .m file (in between the implementation) and then begin to add the function braces and spacing them out:



Now, if you attempt to compile it now (which I do frequently with cmd+b to essentially check for spelling errors) it will compile, but you'll get several warning about "control reaches end of non-void function". That's because we say it's returning something, but in the function body, we don't have it returning anything. That's fine. Go ahead and do this for DXFProtection.m file too.

Now we'll go ahead and fix some of the VERY obvious functions. For example, in DXFProtection, the method "- (BOOL)isAuthorized" is a no-brainer.. it should return YES. So here are some changes I made to DXFProtection:



You can see for demoExpirationDate, I changed the return value to NSDate *. This is just a guess, I haven't verified this yet, but it makes sense to me. Remember, this is just to get started, we may end up doing a complete overhaul on this later 😀

Go ahead and do what you can to DXFActivation.m for now as well.

Here are my four files with the changes made:
http://hotfile.com/dl/67745150/4ffb283/Archive.zip.html

OK. So now we've got the files cleaned up and able to be compiled without error (warnings are okay at this point). Now we need to start fleshing in the functions, so they actually do something! While class-dump will only get the header files, we can use OTX to fill in the implementation details. So go ahead and get the OTX output for the binary.

We'll start with DXFActivation, and work our way through the easy functions first, beginning with +(id)opticsPro6Activation. Here is what we have for that function:

Code:

```
+(id)[DXFActivation opticsPro6Activation]
    +0  00002222  55                      pushl       %ebp
```

```
  +1  00002223  89e5                    movl        %esp,%ebp
  +3  00002225  56                      pushl       %esi
  +4  00002226  53                      pushl       %ebx
  +5  00002227  83ec10                  subl        $0x10,%esp
  +8  0000222a  e800000000              calll       0x0000222f
 +13  0000222f  5b                      popl        %ebx
 +14  00002230  8b83f1fe0f00            movl        0x000ffef1(%ebx),%eax
      alloc
 +20  00002236  89442404                movl        %eax,0x04(%esp)
 +24  0000223a  8b837dff0f00            movl        0x000fff7d(%ebx),%eax
      DXFActivation
 +30  00002240  890424                  movl        %eax,(%esp)
 +33  00002243  e858151000              calll       0x001037a0
      +[DXFActivation alloc]
 +38  00002248  89c6                    movl        %eax,%esi
 +40  0000224a  8b83edfe0f00            movl        0x000ffeed(%ebx),%eax
      opticsPro6Protection
 +46  00002250  89442404                movl        %eax,0x04(%esp)
 +50  00002254  8b8381ff0f00            movl        0x000fff81(%ebx),%eax
      DXFProtection
 +56  0000225a  890424                  movl        %eax,(%esp)
 +59  0000225d  e83e151000              calll       0x001037a0
      +[DXFProtection opticsPro6Protection]
 +64  00002262  89442408                movl        %eax,0x08(%esp)
 +68  00002266  8b83e9fe0f00            movl        0x000ffee9(%ebx),%eax
      initWithProtection:
 +74  0000226c  89442404                movl        %eax,0x04(%esp)
 +78  00002270  893424                  movl        %esi,(%esp)
 +81  00002273  e828151000              calll       0x001037a0
      -[(%esp,1) initWithProtection:]
 +86  00002278  8b93e5fe0f00            movl        0x000ffee5(%ebx),%edx
      autorelease
 +92  0000227e  89550c                  movl        %edx,0x0c(%ebp)
 +95  00002281  894508                  movl        %eax,0x08(%ebp)
 +98  00002284  83c410                  addl        $0x10,%esp
+101  00002287  5b                      popl        %ebx
+102  00002288  5e                      popl        %esi
+103  00002289  c9                      leave
```

```
  +104  0000228a  e911151000              jmp         0x001037a0
        _objc_msgSend
```

which equates to the following ObjC code:
return [[[DXFActivation alloc] initWithProtection:[DXFProtection opticsPro6Protection]] autorelease];

Just by looking at the ObjC string names on the right was I able to determine that. You'll see that the same thing is for +(id)filmPack1Activation:
return [[[DXFActivation alloc] initWithProtection:[DXFProtection filmPack1Protection]] autorelease];

So now we need to figure out the -(id)initWithProtection:(id)arg1 function... Well, from the previous methods we reversed, we know that the argument passed here is a DXFProtection *, so we can change the signature to read:
- (id)initWithProtection:(DXFProtection *)protection
and also make sure to import "DXFProtection.h" file at the top of the DXFActivation.m file

So now here's the initWithProtection method:

Code:

```
-(id)[DXFActivation initWithProtection:]
    +0  00002309  55                    pushl       %ebp
    +1  0000230a  89e5                  movl        %esp,%ebp
    +3  0000230c  56                    pushl       %esi
    +4  0000230d  53                    pushl       %ebx
    +5  0000230e  83ec20                subl        $0x20,%esp
    +8  00002311  e800000000            calll       0x00002316
   +13  00002316  5b                    popl        %ebx
   +14  00002317  8b4508                movl        0x08(%ebp),%eax
   +17  0000231a  8945f0                movl        %eax,0xf0(%ebp)
   +20  0000231d  8d83eafe0f00          leal        0x000ffeea(%ebx),%eax
        DXFActivation
   +26  00002323  8b4004                movl        0x04(%eax),%eax
   +29  00002326  8945f4                movl        %eax,0xf4(%ebp)
   +32  00002329  8d55f0                leal        0xf0(%ebp),%edx
   +35  0000232c  8b831afe0f00          movl        0x000ffe1a(%ebx),%eax
        init
   +41  00002332  89442404              movl        %eax,0x04(%esp)
   +45  00002336  891424                movl        %edx,(%esp)
   +48  00002339  e867141000            calll       0x001037a5
        -[[(%esp,1) super] init]
```

```
 +53  0000233e  89c6                    movl        %eax,%esi
 +55  00002340  85c0                    testl       %eax,%eax
 +57  00002342  7447                    je          0x0000238b
     return;
 +59  00002344  8b8316fe0f00            movl        0x000ffe16(%ebx),%eax
     retain
 +65  0000234a  89442404                movl        %eax,0x04(%esp)
 +69  0000234e  8b4510                  movl        0x10(%ebp),%eax
 +72  00002351  890424                  movl        %eax,(%esp)
 +75  00002354  e847141000              calll       0x001037a0
     -[(%esp,1) retain]
 +80  00002359  894604                  movl        %eax,0x04(%esi)
 +83  0000235c  85c0                    testl       %eax,%eax
 +85  0000235e  7524                    jne         0x00002384
     return;
 +87  00002360  8d83569d0e00            leal        0x000e9d56(%ebx),%eax
     *** error: could not open PACE protection
 +93  00002366  890424                  movl        %eax,(%esp)
 +96  00002369  e83f111000              calll       0x001034ad
     _NSLog
+101  0000236e  8b8312fe0f00            movl        0x000ffe12(%ebx),%eax
     dealloc
+107  00002374  89442404                movl        %eax,0x04(%esp)
+111  00002378  893424                  movl        %esi,(%esp)
+114  0000237b  e820141000              calll       0x001037a0
     -[(%esp,1) dealloc]
+119  00002380  31f6                    xorl        %esi,%esi
+121  00002382  eb07                    jmp         0x0000238b
     return;
+123  00002384  c7461c00000000          movl        $0x00000000,0x1c(%esi)
+130  0000238b  89f0                    movl        %esi,%eax
+132  0000238d  83c420                  addl        $0x20,%esp
+135  00002390  5b                      popl        %ebx
+136  00002391  5e                      popl        %esi
+137  00002392  c9                      leave
+138  00002393  c3                      ret
```

You can see it calls [super init], and actually every ObjC classes init function does this. Then you can see this:

Code:

```
   +69  0000234e  8b4510                  movl        0x10(%ebp),%eax
   +72  00002351  890424                  movl        %eax,(%esp)
   +75  00002354  e847141000              calll       0x001037a0
        -[(%esp,1) retain]
```

so at [ebp+0x10], it's getting passed a retain message. So a quick intro to the stack... [ebp+0x10] actually refers to the first argument passed into this function, the DXFProtection *. Let's think about why... Before a function call, you'll notice that things are assigned to [esp], [esp+4], [esp+8], and so on. So before the call is made, esp points to the first argument, esp+4 points to the second argument, esp+8 points to the third argument, and so on. The assembly "call" instruction actually does 2 things: pushes the next instruction address onto the stack, and then jmps to the address that is being called. So at the first instruction in the call, the stack now points to the return address, and esp-4 is the first argument. Here's my notes from here on further, for the [DXFActivation initWithProtection] method:

pushl %ebp <--- push ebp on stack, stack now points to ebp's value, and [esp - 4] is the return address, [esp - 8] is first argument
movl %esp,%ebp <--- esp and ebp are now equal... ebp points to ebp old value, [ebp - 4] is return address, [ebp - 8] is first argument

So, you can see that [ebp - 8] is first argument, [ebp - 12] (or [ebp - 0xC]) is second argument, [ebp - 16] (or [ebp - 0x10]) is third argument, and so on. Now, behind the scenes, an objective-c method call is actually a C function:
objc_msgSend(id instance, SEL selector, arg1, arg2);
where "instance" is the class instance pointer (in this case, DXFActivation), selector is the function string (in this case "initWithProtection:"), and arg1 is the first argument (DXFProtection *protection, in this case). So you can see that the first ObjC argument is actually the 3rd in the C function... We know that [ebp-8] is arg1, [ebp-0xc] is arg2, and [ebp-0x10] is arg3, so the first argument to an objective-c function is actually at ebp-10.
Hope you could follow that

So it sends argument1 a retain message, which increments the retain count and returns the same object. Immediately afterwards, we see this:

Code:

```
   +75  00002354  e847141000              calll       0x001037a0
        -[(%esp,1) retain]
   +80  00002359  894604                  movl        %eax,0x04(%esi)
   +83  0000235c  85c0                    testl       %eax,%eax
   +85  0000235e  7524                    jne         0x00002384
        return;
   +87  00002360  8d83569d0e00            leal        0x000e9d56(%ebx),%eax
        *** error: could not open PACE protection
   +93  00002366  890424                  movl        %eax,(%esp)
   +96  00002369  e83f111000              calll       0x001034ad
        _NSLog
```

with the "test eax, eax", it's checking to see if the object is 0, or NULL, or nil, and if it is, creates an error, and eventually deallocates the DXFActivation object and returns nil. But before that, it's assigned to [esi+4]...

Code:

```
   +80  00002359  894604                  movl        %eax,0x04(%esi)
```

looking over the function, we see that here:

Code:

```
   +48  00002339  e867141000              calll       0x001037a5
        -[[(%esp,1) super] init]
   +53  0000233e  89c6                    movl        %eax,%esi
```

after the [super init] call, esi was assigned the result. So esi is the current DXFActivation instance. So if we look at the header that class-dump dumped, we see the ivars are:

Code:

```
    DXFProtection *_protection;            <---- [esi+4]
    ActivationWebService *_activationWS;        <---- [esi+8]
    id _delegate;                <---- [esi+12]
    ActivationProgressController *_progressController;    <---- [esi+16]
... etc
```

So it's assigning the first argument, the DXFProtection *, to the ivar _protection, after sending it a retain message.

OK, with this info, here is what I got of the initWithProtection: method:

Code:

```
- (id)initWithProtection:(DXFProtection *)protection
{
   if (!(self = [super init])) return nil;

   if (!protection) {   [self release]; return nil;   }

   _protection = [protection retain];

   return self;
}
```

Now, the +(id)licenseDirectory function is going to require a bit of research...

Here is the OTX output:

Code:

```
+(id)[DXFActivation licenseDirectory]
    +0  00002adb  55                    pushl       %ebp
    +1  00002adc  89e5                  movl        %esp,%ebp
    +3  00002ade  56                    pushl       %esi
    +4  00002adf  53                    pushl       %ebx
    +5  00002ae0  83ec10                subl        $0x10,%esp
    +8  00002ae3  e800000000            calll       0x00002ae8
   +13  00002ae8  5b                    popl        %ebx
   +14  00002ae9  c744240801000000      movl        $0x00000001,0x08(%esp)
   +22  00002af1  c744240402000000      movl        $0x00000002,0x04(%esp)
   +30  00002af9  c704240e000000        movl        $0x0000000e,(%esp)
   +37  00002b00  e8b2091000            calll       0x001034b7
        _NSSearchPathForDirectoriesInDomains
   +42  00002b05  8b9320f60f00          movl        0x000ff620(%ebx),%edx
```

```
      lastObject
  +48  00002b0b  89542404                movl        %edx,0x04(%esp)
  +52  00002b0f  890424                  movl        %eax,(%esp)
  +55  00002b12  e8890c1000              calll       0x001037a0
      -[(%esp,1) lastObject]
  +60  00002b17  8bb31cf60f00            movl        0x000ff61c(%ebx),%esi
      stringByAppendingPathComponent:
  +66  00002b1d  8d9324960e00            leal        0x000e9624(%ebx),%edx
      PACE Anti-Piracy
  +72  00002b23  89542408                movl        %edx,0x08(%esp)
  +76  00002b27  89742404                movl        %esi,0x04(%esp)
  +80  00002b2b  890424                  movl        %eax,(%esp)
  +83  00002b2e  e86d0c1000              calll       0x001037a0
      -[(%esp,1) stringByAppendingPathComponent:]
  +88  00002b33  8d9334960e00            leal        0x000e9634(%ebx),%edx
      License Files
  +94  00002b39  89542408                movl        %edx,0x08(%esp)
  +98  00002b3d  89742404                movl        %esi,0x04(%esp)
 +102  00002b41  890424                  movl        %eax,(%esp)
 +105  00002b44  e8570c1000              calll       0x001037a0
      -[(%esp,1) stringByAppendingPathComponent:]
 +110  00002b49  83c410                  addl        $0x10,%esp
 +113  00002b4c  5b                     popl        %ebx
 +114  00002b4d  5e                     popl        %esi
 +115  00002b4e  c9                     leave
 +116  00002b4f  c3                     ret
```

It's easy to see that it gets a string from NSSearchPathForDirectoriesInDomains... and then appends "PACE Anti-Piracy" to it, and then appends "License Files" to that, for something ending up like this: /path/to/somedir/PACE Anti-Piracy/License Files/.

So NSSearchPathForDirectoriesInDomains... we need to find out what that returns. Here are the arguments passed to it:

Code:

```
   +14  00002ae9  c744240801000000        movl        $0x00000001,0x08(%esp)
   +22  00002af1  c744240402000000        movl        $0x00000002,0x04(%esp)
   +30  00002af9  c704240e000000          movl        $0x0000000e,(%esp)
   +37  00002b00  e8b2091000              calll       0x001034b7
        _NSSearchPathForDirectoriesInDomains
```

or more readably
NSSearchPathForDirectoriesInDomains(0xE, 0x2, 0x1)

The arguments are probably enums... lets google NSSearchPathForDirectoriesInDomains.

First link in Google: Foundations Functions Reference
http://developer.apple.com/mac/library/ ... rence.html

Looking at the documentation for this methods, we can see the function signature is
NSArray * NSSearchPathForDirectoriesInDomains (
NSSearchPathDirectory directory,
NSSearchPathDomainMask domainMask,
BOOL expandTilde
);

now if you search for NSSearchPathDirectory and NSSearchPathDomainMask in google, you'll find these references:

Code:

```
NSSearchPathDirectory
These constants specify the location of a variety of directories.

enum {
   NSApplicationDirectory = 1,
   NSDemoApplicationDirectory,
   NSDeveloperApplicationDirectory,
   NSAdminApplicationDirectory,
   NSLibraryDirectory,
   NSDeveloperDirectory,
   NSUserDirectory,
```

```
   NSDocumentationDirectory,
   NSDocumentDirectory,
   NSCoreServiceDirectory,
   NSAutosavedInformationDirectory = 11,
   NSDesktopDirectory = 12,
   NSCachesDirectory = 13,
   NSApplicationSupportDirectory = 14,
   NSDownloadsDirectory = 15,
   NSInputMethodsDirectory = 16,
   NSMoviesDirectory = 17,
   NSMusicDirectory = 18,
   NSPicturesDirectory = 19,
   NSPrinterDescriptionDirectory = 20,
   NSSharedPublicDirectory = 21,
   NSPreferencePanesDirectory = 22,
   NSItemReplacementDirectory = 99,
   NSAllApplicationsDirectory = 100,
   NSAllLibrariesDirectory = 101
};
typedef NSUInteger NSSearchPathDirectory;
```

Code:

```
NSSearchPathDomainMask
Search path domain constants specifying base locations for the
NSSearchPathDirectory type.

enum {
   NSUserDomainMask = 1,
   NSLocalDomainMask = 2,
   NSNetworkDomainMask = 4,
   NSSystemDomainMask = 8,
   NSAllDomainsMask = 0x0ffff,
};
typedef NSUInteger NSSearchPathDomainMask;
```

So my reversal of the function is:

Code:

```
+ (NSString *)licenseDirectory
{
   NSArray *directories =
NSSearchPathForDirectoriesInDomains(NSApplicationSupportDirectory,
NSLocalDomainMask, YES);
   return [[[directories lastObject] stringByAppendingPathComponent:@"PACE
Anti-Piracy"] stringByAppendingPathComponent:@"License Files"];
}
```

We probably will be eliminating this function, but I think it's a good exercise to show you how to reverse objective-c functions from OTX back into compilable source code.

Go ahead and continue on reversing each function for and filling in the methods you have in your DXFActivation and DXFProtection files. For reference, once you are complete, here is what I came up with for DXFActivation:
http://hotfile.com/dl/67765165/b6a5cd8/Archive.zip.html

So we can see that the DXFActivation class is essentially a wrapper around the DXFProtection class, as each call made to DXFActivation just calls a related function in the DXFProtection class it was initialized with. So it seems the protection lies in, duh, the DXFProtection class. Now, we have complete control over what these functions return, so our job here is to understand them and see which ones we need to change, and then change them to return the appropriate value.

Titan 4.0.6 Reversing Notes

The_maniacz MSJ

After stripping the pace and launching, it will crash. On looking at the crash report, you will find this:

Code:

```
Thread 0 Crashed:
0   ???                                 0x003f1478 0 + 4134008
1   ???                                 0x003f1ef1 0 + 4136689
2   com.SynchroArts.Titan4              0x00013896 start + 166
3   com.SynchroArts.Titan4              0x00013819 start + 41
```

If you look at the OTX output at 0x00013896 , it will be as follows:

Code:

```
+101  0001387f  a108a03e00              movl        0x003ea008,%eax
  +106  00013884  8b00                    movl        (%eax),%eax
  +108  00013886  85c0                    testl       %eax,%eax
  +110  00013888  7402                    je          0x0001388c
  +112  0001388a  ffd0                    call        *%eax
  +114  0001388c  e84a723d00              calll       0x003eaadb
 ___keymgr_dwarf2_register_sections
  +119  00013891  e844e63d00              calll       0x003f1eda
  +124  00013896  eb0f                    jmp         0x000138a7
```

Here, a call to 0x003eaadb will cause a Kernel panic. Looking back, there is a jump if equal to 0x0001388c at 0x00013888.
I did a breakpoint in gdb at 0x00013888.
There, I caused it to Jump to 0x0001388c instead of Jump if equal. That caused a crash.

Code:

```
  +110  00013888  7402                    je          0x0001388c
  +112  0001388a  ffd0                    call        *%eax
  +114  0001388c  e84a723d00              calll       0x003eaadb          ___
keymgr_dwarf2_register_sections
  +119  00013891  e844e63d00              calll       0x003f1eda
  +124  00013896  eb0f                    jmp         0x000138a7
```

```
  +126  00013898  89bb1e004efc            movl        %edi,0xfc4e001e(%ebx)
  +132  0001389e  100e                   adcb        %cl,(%esi)
```

The reason is it's again calling to a 0x0003 address. So instead of jump to 0x0001388c, I did Jump to 0x000138a7 which is after those two calls to a 0x003 address (at +114 and +119)

That jump too caused a crash because following that code is :

Code:

```
  +141  000138a7  8d45e4                 leal        0xe4(%ebp),%eax
  +144  000138aa  89442404               movl        %eax,0x04(%esp)
  +148  000138ae  c7042464062000         movl        $0x00200664,(%esp)
      __dyld_mod_term_funcs
  +155  000138b5  e846000000             calll       0x00013900
  +160  000138ba  8b45e4                 movl        0xe4(%ebp),%eax
  +163  000138bd  85c0                   testl       %eax,%eax
  +165  000138bf  7408                   je          0x000138c9
  +167  000138c1  890424                 movl        %eax,(%esp)
  +170  000138c4  e820020000             calll       0x00013ae9
      Anon478
  +175  000138c9  a104a03e00             movl        0x003ea004,%eax
```

Here at +165, there is a

Code:

```
je          0x000138c9
```

Since it crashed, that wasn't executed. So, I put a Breakpoint at 0x000138bf and once it breaks did a Jump to 0x000138c9.

Voila!! Titan Loaded...

Code:

```
Reading symbols for shared libraries . done

Breakpoint 2, 0x0001381a in start ()
(gdb) c
Continuing.

Breakpoint 3, 0x00013888 in start ()
(gdb) jump *0x000138a7
Continuing at 0x138a7.
```

```
Breakpoint 4, 0x000138bf in start ()
(gdb) jump *0x000138c9
Continuing at 0x138c9.
Reading symbols for shared libraries .. done
Reading symbols for shared libraries . done
Reading symbols for shared libraries . done
Reading symbols for shared libraries . done
Reading symbols for shared libraries . done

Program exited normally.
```

The working method would be to cause a jump to 0x000138c9 at

Code:

```
00013888  7402                    je          0x0001388c
```

instead of 0x0001388c, thereby bypassing the check. That is done in a hex editor by changing `7402` to `EB3F`

Edi Prompt 1.3.0 Crack Notes

The_maniacz MSJ

Unpace Edi Prompt . It will give you an error saying file is not protected by interlok. Now, Load terminal.

Code:

```
cd / path to app
grep -i -r sgemexam .
Binary file
./Contents/Frameworks/pEdiPrompt001.bundle/Contents/MacOS/EdiPrompt001
matches
```

Remember the . after sgemexam. (Thanks to unsanity for this tip!)

That means the file to be unpaced is in *app/ Contents/Frameworks/pEdiPrompt001.bundle/Contents/MacOS/EdiPrompt001*

Unpace this with the -f and -b flag (because it's a bundle). Now, when you load, it will give you an authorization error. Load EdiPrompt in OTX and check the output. As usual, check for Auth or PACE etc. You will come to this:

Code:

```
_AppSIS.CheckPACEAuthedOkay%%o<AppSIS>si4b:
   +0   0055b930  55                pushl      %ebp
   +1   0055b931  8bec                movl     %esp,%ebp
   +3   0055b933  81ec54000000          subl     $0x00000054,%esp
   +9   0055b939  53                pushl      %ebx
```

Aha! You can make that return 1 by going to a hex editor and changing the values of
`558BEC81EC54000000`
to
`31C040C3EC54000000`

This is basically

Code:

```
xorl   %eax,%eax
incl   %eax
ret
```

There it is!! You have done it!

Sysenter break and reversing in Safari

Safari uses the ptrace() syscall to try and stop evil hackers from debugging their proprietary code. ;). The extract from the man-page below shows a ptrace() flag which can be used to stop people being able to debug your code.

PT_DENY_ATTACH
This request is the other operation used by the traced process; it allows a process that is not currently being traced to deny future traces by its parent. All other arguments are ignored. If the process is currently being traced, it will exit with the exit status of ENOTSUP; otherwise, it sets a flag that denies future traces. An attempt by the parent to trace a process which has set this flag will result in a segmentation violation in the parent.
There are a couple of ways to get around this check (which i am aware of).The first of these is to patch your kernel to stop the PT_DENY_ATTACH call from doing anything. This is probably the best way, however involves the most effort.
The method which we will use now to look at Safari is to start up gdb and put a breakpoint on the ptrace() function. This is shown below:

```
-[nemo@gir:~]$ gdb /Applications/Safari.app/Contents/MacOS/Safari
GNU gdb 6.1-20040303 (Apple version gdb-413)
(gdb) break ptrace
Breakpoint 1 at 0x900541f4
```

We then run the program, and wait until the breakpoint is hit. When our breakpoint is triggered, we use the x/10i $pc command (below) to view the next 10 instructions in the function.

```
(gdb) r
Starting program: /Applications/Safari.app/Contents/MacOS/Safari
Reading symbols for shared libraries ..... done


Breakpoint 1, 0x900541f4 in ptrace ()
(gdb) x/10i $pc
0x900541f4 <ptrace+20>: addis   r8,r8,4091
0x900541f8 <ptrace+24>: lwz     r8,7860(r8)
0x900541fc <ptrace+28>: stw     r7,0(r8)
0x90054200 <ptrace+32>: li      r0,26
0x90054204 <ptrace+36>: sc
0x90054208 <ptrace+40>: b       0x90054210 <ptrace+48>
0x9005420c <ptrace+44>: b       0x90054230 <ptrace+80>
0x90054210 <ptrace+48>: mflr    r0
0x90054214 <ptrace+52>: bcl-    20,4*cr7+so,0x90054218
0x90054218 <ptrace+56>: mflr    r12
```

At line 0x90054204 we can see the instruction "sc" being executed. This is the instruction which calls the syscall itself. This is similar to int 0x80 on a Linux platform, or sysenter/int 0x2e in windows.
In order to stop the ptrace() syscall from occurring we can simply replace this instruction in memory with a nop (no operation) instruction. This way the syscall will never take place and we can debug without any problems. To patch this instruction in gdb we can use the command shown below and continue execution.

```
(gdb) set *0x90054204 = 0x60000000
(gdb) continue
```

Reversing Unity 3.0 and additional Pace reversing

By unsanity (MSJ)

I took a look at this and it seems to be pretty interesting. First things first, however, is there's a problem with XVX's unpace tool. When it says that it's "Patching PACE Loader", well, it gets something wrong there. So we're just going to remove that part of it, and trust me, you'll have a better working tool because of it.
So open up unpace in OTX and search for "patching pace loader". Here's what you'll find: (I cleaned it up a little bit so it's easier to read here)

Code:

```
0x4287  c7042450580000          movl        $0x00005850,(%esp)[!] Info
patching PACE loader
0x428e  e88d180000              calll       0x00005b20                  _puts
0x4293  8b9598edffff            movl        0xffffed98(%ebp),%edx
0x4299  c7827704000031c040c3    movl        $0xc340c031,0x00000477(%edx)
0x42a3  83a574ffffffef          andl        $0xef,0xffffff74(%ebp)
0x42aa  be01000000              movl        $0x00000001,%esi
```

So this is where it's patching the "PACE Loader", but it doesn't look like it's doing anything very sophisticated. The interesting instruction is:

Code:

```
movl        $0xc340c031,0x00000477(%edx)
```

where it's moving 31 c0 40 c3 (xor eax, eax)(inc eax)(ret) to some address. So that's all there is to the patch. But after unpacing Unity 3.0 with unpace, it crashes on these instruction because it wrote them in the wrong spot. So here in unpace we're going to nop out these instruction, basically we don't want it to

patch the pace loader anymore, we'll do that ourselves .
So nop out the instruction

movl $0xc340c031,0x00000477(%edx)
You can just nop out all the instructions so the text doesn't even show when you run unpace by nopping out the following instructions:

Code:

```
0x4287  c7042450580000          movl        $0x00005850,(%esp)          [!]
Info: patching PACE loader
0x428e  e88d180000              calll       0x00005b20                _puts
0x4293  8b9598edffff            movl        0xffffed98(%ebp),%edx
0x4299  c7827704000031c040c3    movl        $0xc340c031,0x00000477(%edx)
```

And there! You have a better unpace and we can now unpace Unity with no problems! We'll tackle the license part next. But first we have to patch the pace loader manually. But let's assume we don't know what the pace loader is or what we need to patch.
First, we'll start by running Unity under gdb. First thing we get is the "Program exited with code 055." Which is the standard ptrace pt_deny_attach error. So your going to have to load an antiptrace kext for this one..
Ok once it's loaded, let's try it again. Oops, we get this:

Code:

```
gdb$ r
warning: Trying to remove a section from the ordered section list that did
not exist at 0xe16b000.
Reading symbols for shared libraries
.++++++++++++++++++++++++++++++.........................................
.................................................... done
warning: Unable to read symbols for "[Unknown Bundle]" (file not found).
Reading symbols for shared libraries . done

Program exited with code 0254.
--------------------------------------------------------------------------
-------------[regs]
  EAX:Error while running hook_stop:
No registers.
```

and a fancy error message that says Unity can't be loaded. But hey, look here! We can see the line that says "Unable to read symbols for "[Unknown Bundle]"". Well we know that that is PACE related. So it's trying to load those embedded pace bundles.... Well, I once told you that the entrypoint for [Bundle] is IL_SC_D_PPC, and the entrypoint for [Unknown Bundle] is EntryPoint... So let's go ahead and set a break on EntryPoint and then do a backtrace.

Code:

```
gdb$ hb EntryPoint
gdb$ r
gdb$ bt
#0  0x04800ad3 in EntryPoint ()
#1  0x013eff4b in dyld_stub_writev ()
#2  0x0000353d in _start ()
#3  0x000034b9 in start ()
```

Ahhhh so before _main is ever even called, it's still loading some pace stuff. OK, so now that we know where it's originating from, let's quit and just load Unity up in gdb again and break on start.

Code:

```
gdb$ hb start
gdb$ r
```

You'll see something like this

Code:

```
0x34b0 <start+32>:   mov    DWORD PTR [esp+0x8],ebx
0x34b4 <start+36>:   call   0x34ba <_start>
0x34b9 <start+41>:   hlt
```

Notice how the 0x34b9 here corresponds to the first address in the backtrace, that means we need to follow the call immediately before it, so the call 0x34ba..
So once we follow that call in, we need to look for the next address that's in the backtrace, 0x353d. So step through until we get to that address, and we can see it's here:

Code:

```
0x3531 <_start+119>:   call   eax
0x3533 <_start+121>:   call   0x13e7fe6
<dyld_stub___keymgr_dwarf2_register_sections>
0x3538 <_start+126>:   call   0x13efeda
0x353d <_start+131>:   jmp    0x354e <_start+148>
```

So we need to follow the call in immediately before 0x353d, which is the call to 0x13efeda.

OK, in this call, we're looking for the next address in the backtrace, which was 0x013eff4b. So do

Code:

```
gdb$ x/50i $pc
```

So we can see the instructions for this call. And a little ways down in the output, we see:

Code:

```
0x13eff41:   mov    DWORD PTR [esp],0x0
0x13eff48:   call   DWORD PTR [esi+0x54]
0x13eff4b:   mov    DWORD PTR [esi+0x58],0x0
0x13eff52:   call   0x13f0077
```

So the "call DWORD PTR [esi+0x54]" is obviously the call to EntryPoint, which is not good. That means it's calling PACE. A little bit before that we see

Code:

```
0x13eff17:   call   DWORD PTR [ebp-0x1c]
0x13eff1a:   call   DWORD PTR [ebp-0xc]
0x13eff1d:   mov    DWORD PTR [esp+0x4],edi
0x13eff21:   mov    DWORD PTR [esp],esi
0x13eff24:   call   0x13ef60a
0x13eff29:   test   eax,eax
0x13eff2b:   jne    0x13eff82
```

and that jne, if true, jumps over that pace call. Hmmmm. We remember that thing in unpace that we nopped out was replacing something with 31 c0 40 c3, which is the standard way to force a function to return true. Well, if we force the call to 0x13ef60a to return true, then that jne will jump over the call to EntryPoint. Let's try that out.

Code:

```
gdb$ hexdump 0x13ef60a
013EF60A : 55 89 E5 57  56 83 EC 50 - 8B 7D 08 8B  4F 54 85 C9
U..WV..P.}..OT..
```

We do a hexdump of that call to get the bytes, and it gives us 5589E5575683EC508B7D088B4F5485C9. So we open up Unity in a hex editor, search for that hex string, and replace the beginning with 31 c0 40 c3, save it, and then run Unity!

It seems to work! All we get is a "Unity license invalid" message, which means we're good to move onto the licensing part.
Congratulations, you have manually done what unpace does!

If you look, you will find this in gdb:

0x005b6170 <_ZN11LicenseInfo20InitializeProtectionEv+0>: push %ebp
simply return 1 and it works. But about screen shows unregistered. You see, all throughout the app, when it wants to check the license info, it calls LicenseInfo::Get(). In this function, it checks to see if the license info has been set up yet; if it has, it returns it, otherwise it creates it and returns it. When it creates it the first time, it calls LicenseInfo::InitializeProtection... it is this function that is vulnerable, because it passes in the newly allocated license structure as arg2. There is a BUNCH of space in this function, and it's whole purpose is to set up the license structure that was passed in. So what I did was this:

Copy-paste the following into a blank textedit document (be sure to set it to plain-text... Format->Make Plain Text)

Code:

```
BITS 32
mov DWORD [esi],    0x0
mov DWORD [esi+4],  0x0
mov BYTE  [esi+4],  0x1
```

```
mov DWORD [esi+8],  0x100
mov DWORD [esi+12], 0xFFFFFFFF
mov DWORD [esi+16], 0x40000000
mov DWORD [esi+20], 0x542d3355
mov DWORD [esi+24], 0x2d4b424b
mov DWORD [esi+28], 0x35383334
mov DWORD [esi+32], 0x4656512d
mov DWORD [esi+36], 0x33382d57
mov DWORD [esi+40], 0x592d3736
mov DWORD [esi+44], 0x00363545
mov DWORD [esi+48], 0x0
mov DWORD [esi+52], 0x1000000
add esp, 0xBC
pop ebx
pop esi
pop edi
pop ebp
ret
```

Then save it to your desktop as "license.s". This is some assembly code that we're going to compile and replace the function InitializeProtection with. Now open up terminal and type

Code:

```
nasm -f bin -o ~/Desktop/UnityLicenseASM ~/Desktop/license.s
```

This command will compile the assembly in the license.s file to a binary file on your desktop called UnityLicenseASM. Now, open up this binary in 0xED hex editor, and also open up the Unity binary in 0xED as well. In the UnityLicenseASM document, press cmd+A to , then copy it. In the Unity document, go to offset 0x5b517f and paste in the bytes you copied from UnityLicenseASM. Now save it and you're done!

This is what that function was before:

Code:

```
LicenseInfo::InitializeProtection()
    +0  005b6170  55                    pushl       %ebp
    +1  005b6171  89e5                  movl        %esp,%ebp
    +3  005b6173  57                    pushl       %edi
    +4  005b6174  56                    pushl       %esi
    +5  005b6175  53                    pushl       %ebx
    +6  005b6176  81ecbc000000          subl        $0x000000bc,%esp
   +12  005b617c  8b7508                movl        0x08(%ebp),%esi
   +15  005b617f  c704241427ed00        movl        $0x00ed2714,(%esp)
       LicenseInfo:InitializeProtection\n
   +22  005b6186  e835eac6ff            calll       0x100224bc0
```

```
   +27  005b618b  8d45c4                  leal        0xc4(%ebp),%eax
   +30  005b618e  c6461400                movb        $0x00,0x14(%esi)
   +34  005b6192  c7463401000000          movl        $0x00000001,0x34(%esi)
   +41  005b6199  894588                  movl        %eax,0x88(%ebp)
   +44  005b619c  890424                  movl        %eax,(%esp)
   +47  005b619f  e81cbf1f00              calll       GetCocoaCurrentDirector
y()
   +52  005b61a4  c7460c00000000          movl        $0x00000000,0x0c(%esi)
   +59  005b61ab  c7461000000000          movl        $0x00000000,0x10(%esi)
   +66  005b61b2  83ec04                  subl        $0x04,%esp
   +69  005b61b5  807e0400                cmpb        $0x00,0x04(%esi)
   +73  005b61b9  740a                    je          0x005b61c5
   +75  005b61bb  8b06                    movl        (%esi),%eax
   +77  005b61bd  85c0                    testl       %eax,%eax
   +79  005b61bf  0f852b040000            jne         0x005b65f0
   +85  005b61c5  8b4588                  movl        0x88(%ebp),%eax
   +88  005b61c8  8d5dc8                  leal        0xc8(%ebp),%ebx
   +91  005b61cb  891c24                  movl        %ebx,(%esp)
   +94  005b61ce  89442404                movl        %eax,0x04(%esp)
   +98  005b61d2  e8931ce300              calll       0x013e7e6a
        std::basic_string<char, std::char_traits<char>, std::allocator<char>
>::basic_string(std::basic_string<char, std::char_traits<char>,
std::allocator<char> > const&)
  +103  005b61d7  891c24                  movl        %ebx,(%esp)
  +106  005b61da  e891be1f00              calll       SetCocoaCurrentDirector
y(std::basic_string<char, std::char_traits<char>, std::allocator<char> >)
  +111  005b61df  891c24                  movl        %ebx,(%esp)
  +114  005b61e2  e8a11ce300              calll       0x013e7e88
        std::basic_string<char, std::char_traits<char>, std::allocator<char>
>::~basic_string()
  +119  005b61e7  c70600000000            movl        $0x00000000,(%esi)
  +125  005b61ed  c6460400                movb        $0x00,0x04(%esi)
  +129  005b61f1  c704245827ed00          movl        $0x00ed2758,(%esp)
         open PACE from our own executable\n
  +136  005b61f8  e8c3e9c6ff              calll       0x100224bc0
  +141  005b61fd  e89cd4e200              calll       0x013e369e
        _CFBundleGetMainBundle
```

And this is what it is now:

Code:

```
LicenseInfo::InitializeProtection()
    +0  005b6170  55                      pushl       %ebp
    +1  005b6171  89e5                    movl        %esp,%ebp
    +3  005b6173  57                      pushl       %edi
    +4  005b6174  56                      pushl       %esi
```

```
   +5  005b6175  53                    pushl       %ebx
   +6  005b6176  81ecbc000000          subl        $0x000000bc,%esp
  +12  005b617c  8b7508                movl        0x08(%ebp),%esi
  +15  005b617f  c70600000000          movl        $0x00000000,(%esi)
  +21  005b6185  c7460400000000        movl        $0x00000000,0x04(%esi)
  +28  005b618c  c6460401              movb        $0x01,0x04(%esi)
  +32  005b6190  c7460800010000        movl        $0x00000100,0x08(%esi)
  +39  005b6197  c7460cffffffff        movl        $0xffffffff,0x0c(%esi)
  +46  005b619e  c7461000000040        movl        $0x40000000,0x10(%esi)
  +53  005b61a5  c7461455332d54        movl        $0x542d3355,0x14(%esi)
     'T-3U'
  +60  005b61ac  c746184b424b2d        movl        $0x2d4b424b,0x18(%esi)
     '-KBK'
  +67  005b61b3  c7461c34333835        movl        $0x35383334,0x1c(%esi)
     '5834'
  +74  005b61ba  c746202d515646        movl        $0x4656512d,0x20(%esi)
     'FVQ-'
  +81  005b61c1  c74624572d3833        movl        $0x33382d57,0x24(%esi)
     '38-W'
  +88  005b61c8  c7462836372d59        movl        $0x592d3736,0x28(%esi)
     'Y-76'
  +95  005b61cf  c7462c45353600        movl        $0x00363545,0x2c(%esi)
 +102  005b61d6  c7463000000000        movl        $0x00000000,0x30(%esi)
 +109  005b61dd  c7463400000001        movl        $0x01000000,0x34(%esi)
 +116  005b61e4  81c4bc000000          addl        $0x000000bc,%esp
 +122  005b61ea  5b                    popl        %ebx
 +123  005b61eb  5e                    popl        %esi
 +124  005b61ec  5f                    popl        %edi
 +125  005b61ed  5d                    popl        %ebp
 +126  005b61ee  c3                    ret
```

You can see why I put in the add 0xbc, esp and the pop ebx, esi, etc... at the end... It was to balance out the prolog in the beginning. Basically what the assembly code we put in there does is set up the license structure to a valid pro license, with no restrictions. Remember, The serial is unimportant. Try replacing

those with anything You can even make them all "X"'s if you want, it does not matter. I just used something from the top of my head.

Reversing Protools 9.0 HD

By unsanity (MSJ)

Today, I will begin showing you how to crack Pro Tools 9. It uses the new protection by PACE, Eden, as well as some advanced checksumming features which appear to be part of the new PACE SDK and which prevent any modification to the protected binary. This seems to be the only addition to the new PACE 2 protection: a license server app and inline checksumming. Other than that, the prot has not changed from Pro Tools 8.

Tools Required:

UnLOK 2.1 http://www.mediafire.com/?3hbada9lrtzqyy1 (10.6+ only)

Pro Tools 9 Installer

IDA Pro 5.5 (or 6.0 if you're lucky enough)

GDB

Your life will be easier with an antiptrace kext. This kext is only for 10.6.4, you will kernel panic attempting to load it on any other version.

http://www.mediafire.com/?o46lu4k692ux63h

Using antiptrace.kext:

Extract the zip file and the antidebug.kext to your desktop.
open terminal
type the following to install

Code:

```
sudo cp -R ~/Desktop/antidebug.kext /System/Library/Extensions/
```

type the following to load:

Code:

```
sudo kextload /System/Library/Extensions/antidebug.kext
```

to modify attributes of the kext:

Code:

```
sysctl -w debug.antidebug.ptrace=(0 or 1)
sysctl -w debug.antidebug.sysctlantidebug=(0 or 1)
sysctl -w debug.antidebug.extendedattr=(0 or 1)
sysctl -w debug.antidebug.dumpextendedattr=(0 or 1)
```

Step 1: The Frameworks

or, ***I Wasn't Sure What to PACE, So I Just PACE'd Everything!***

Go ahead and install Pro Tools 9.

Now for the next part. So for me, this was the most annoying part. As we know from similar protections in the past, more than just the binary is protected here. If you recall, you can find this out by grep'ing for the magic bytes **sgemexam**.

Code:

```
grep -i -r sgemexam /path/to/protools.app
```

and

Code:

```
grep -i -r sgemexam /Library/Frameworks
```

This will list all the binaries protected with PACE.

I was originally going to tell you to use the dummy framework trick to unpace the frameworks, but instead I posted UnLOK in my previous post. Download it, and run it on all the frameworks to unpace them. I'm not sure if it works completely accurately on the actual binary, but use either it or XVX's unpace to unpace it.

Hint, when UnLOK'ing the framework, be sure to drag the actual binary (located in /Versions/A) to the UnLOK window, or else it won't work. Well, it works, but UnLOK will replace the symlink located in the base framework directory instead. Of course, you could just move that one into /Versions/A and create a new symlink... but better to do it right from the get-go.

And that's it for the frameworks!

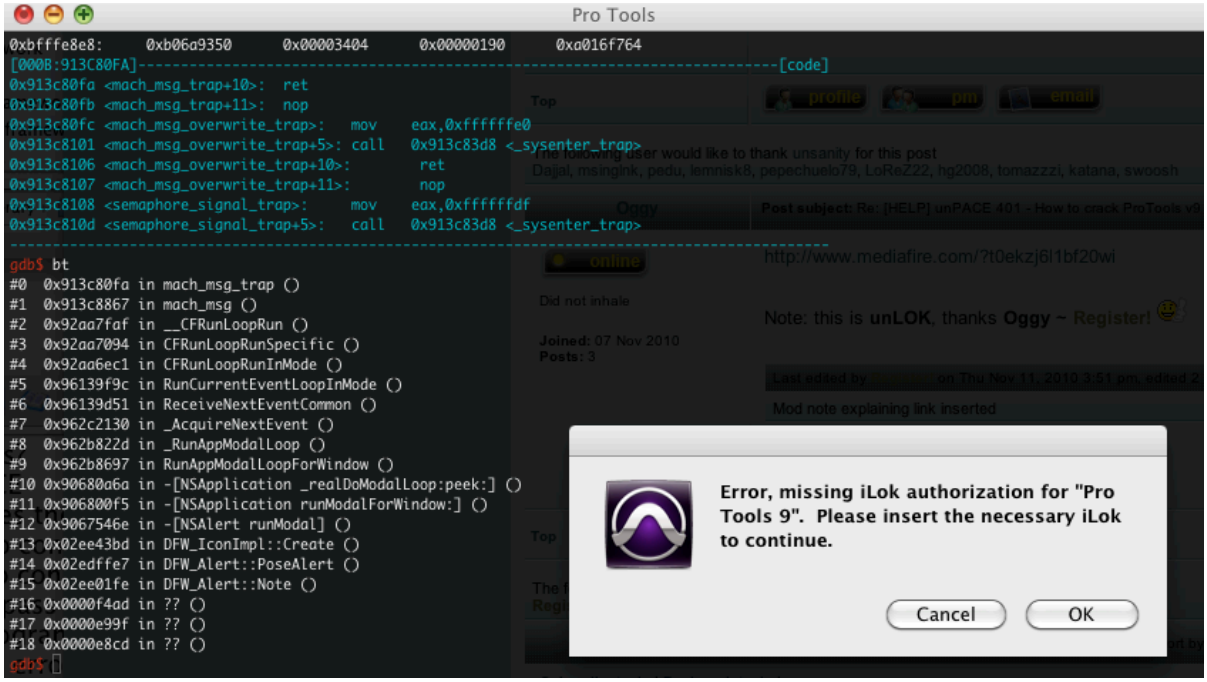
Step 2: Reconnaissance

or, ***Playing in the Mud***

Next step is to actually find out how the protection implements itself. Go ahead and start Pro Tools under gdb and let's see how far we get... Not very.

We get a dialog box first thing saying that there's no ilok auth. OK, so where is that called from in the

code? Looks like the dialog box is running modally. hmmm. Press ctrl+c in the gdb window and do a backtrace...



Well this looks very interesting... DFW_Alert::Note ()
Now that sounds like it's supposed to display an alert box, doesn't it? So what we're interested in is what function called the alert box, because there was a decision that had to be made somewhere in there to decide whether the program was authorized or not, and therefore whether to display the dialog or not. So we are interested in the address right before DFW_Alert::Note (). That address is 0xF4AD.

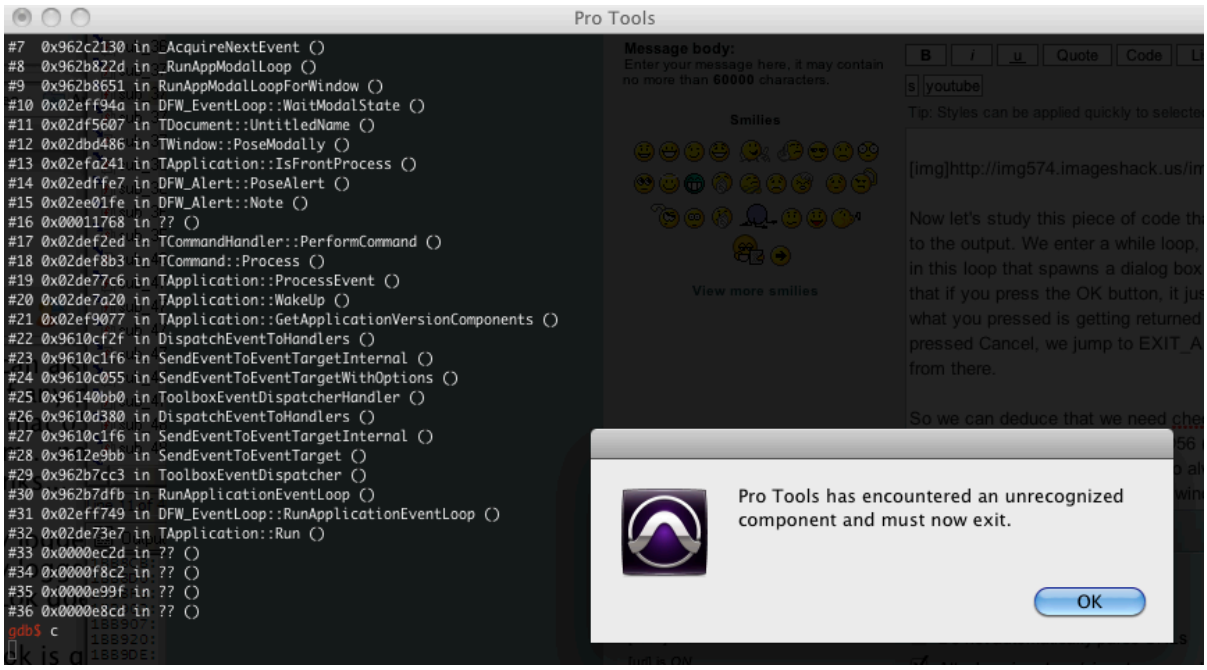
Now what we know about that address is that it's pointing to the NEXT instruction after the call the DFW_Alert::Note(). Remember, that's how back traces work. So fire up IDA Pro and disassemble the main binary, looking for address 0xF4AD. Once you find it, go ahead and press F5 to enter decompilation mode, and we can see that we end up right here:

```
  &TApplication__sPatentString,
  "Pro Tools-« or portions thereof are protected by the following United States Patent: 6,704,671.
if ( SPSStartupOptionSInt32__operator long(&unk_1B7CCC) > 0 )
{
  v12 = SPSStartupOptionSInt32__operator long(&unk_1B7CCC);
  sub_11D6D(v12);
}
alerrtReturn = 0;
while ( !checkIlokAuth(0) )
{
  sub_FE74(&v33);
  sub_FE40(&messageText);
  alerrtReturn = DFW_Alert__Note(&messageText, &DFW_Alert__sOKCancel, 0, 0, 1, -1, 0, 0);
  std__string___Rep___M_dispose(messageText - 12);
  std__string___Rep___M_dispose(v33 - 12);
  if ( alerrtReturn )
    goto EXIT_APP;
}
if ( !alerrtReturn )
{
  sub_EE3E();
  sub_11C96();
```

Now let's study this piece of code that calls the alert... As you can see, I already made some changes to the output. We enter a while loop, which means that as long as checkIlokAuth returns 0, we will stay in this loop that spawns a dialog box. Now, when the dialog box comes up on your screen, you'll see that if you press the OK button, it just comes back again, but if you press Cancel, the app quits. So what you pressed is getting returned back in alertReturn (which is 0 for OK, and 1 for Cancel). If you pressed Cancel, we jump to EXIT_APP, but if you press OK, then we do checkIlokAuth again and go from there.

So we can deduce that we need checkIlokAuth (sub_49D56) to return 1 to avoid that error. OK, so open up OxED and patch offset 0x48D56 (yes, remember to account for vm_addr_slide, or the fact that the binary is mapped in at 0x1000, so always remember to subtract 0x1000 from the address if necessary) to 31 C0 40 C3... And no more Ilok window!

But now we end up with this when we ctrl+c and backtrace on the next window We didn't think it would be that easy, did we?


Pro Tools
#7 0x962c2130 in _AcquireNextEvent ()
#8 0x962b822d in _RunAppModalLoop ()
#9 0x962b8651 in RunAppModalLoopForWindow ()
#10 0x02eff94a in DFW_EventLoop::WaitModalState ()
#11 0x02df5607 in TDocument::UntitledName ()
#12 0x02dbd486 in TWindow::PoseModally ()
#13 0x02efa241 in TApplication::IsFrontProcess ()
#14 0x02edffe7 in DFW_Alert::PoseAlert ()
#15 0x02ee01fe in DFW_Alert::Note ()
#16 0x00011768 in ?? ()
#17 0x02def2ed in TCommandHandler::PerformCommand ()
#18 0x02def8b3 in TCommand::Process ()
#19 0x02de77c6 in TApplication::ProcessEvent ()
#20 0x02de7a20 in TApplication::WakeUp ()
#21 0x02ef9077 in TApplication::GetApplicationVersionComponents ()
#22 0x9610cf2f in DispatchEventToHandlers ()
#23 0x9610c1f6 in SendEventToEventTargetInternal ()
#24 0x9610c055 in SendEventToEventTargetWithOptions ()
#25 0x96140bb0 in ToolboxEventDispatcherHandler ()
#26 0x9610d380 in DispatchEventToHandlers ()
#27 0x9610c1f6 in SendEventToEventTargetInternal ()
#28 0x9612e9bb in SendEventToEventTarget ()
#29 0x962b7cc3 in ToolboxEventDispatcher ()
#30 0x962b7dfb in RunApplicationEventLoop ()
#31 0x02eff749 in DFW_EventLoop::RunApplicationEventLoop ()
#32 0x02de73e7 in TApplication::Run ()
#33 0x0000ec2d in ?? ()
#34 0x0000f8c2 in ?? ()
#35 0x0000e99f in ?? ()
#36 0x0000e8cd in ?? ()
gdb$ c
Message body:
Enter your message here, it may contain no more than 60000 characters.
Smilies
View more smilies
Pro Tools has encountered an unrecognized component and must now exit.
OK

Which is not much better. The backtrace shows that the app has already entered an event loop of some kind, and is now processing commands. This is very hard to debug, because this dialog is a result of a command being queued in the event loop for processing AFTER the call stack is no longer useful. So it failed a check somewhere, and allocated a QuitCommand or something, sent it to the event queue processor, and the program quits with an error message. Now, I'm going to show you a very advanced trick that not many people know, so learn it, and remember it!

Quit out of gdb back to the command prompt. There's an environment variable you can set that will have the OS record malloc calls, which as you can imagine are responsible for object allocations (like this command that's being processed here). It's MallocStackLogging, and you'll want to set it to 1, like this:

Code:

```
export MallocStackLogging=1
```

Now go ahead and start gdb again. You'll notice some extra output at the beginning, stuff like this (look for it to verify that you have set the variable correctly):

Code:

```
sh(14557) malloc: recording malloc stacks to disk using standard recorder
sh(14557) malloc: stack logs being written into /tmp/stack-
logs.14557.sh.PXhaIm.index
sh(14558) malloc: stack logs deleted from /tmp/stack-
logs.14557.sh.PXhaIm.index
```

OK, now that we are logging malloc calls, go ahead and look at the backtrace from before. Doesn't it make sense that the function TCommand::Process () would be responsible for processing all the commands (of type TCommand)? So let's set a breakpoint there and see what argument/command was passed to that function.

Code:

```
hb TCommand::Process()
```

then go ahead and run gdb

We break on TCommand::Process(), after the function prolog it seems. Now, if you remember, once the function prolog completes, argument one is at [ebp+8], and argument 2 is at [ebp+12], etc... So we need to find out what that argument is....

I've created a very simple command that you can include in your gdbinit file that will make this a piece of cake. So go ahead and open your gdbinit file in a text editor, and find a good spot to put this next script:

Code:

```
define cppclassname
    if $argc == 0
        help cppclassname
    end
    if $argc == 1
        x/s *(int *)((*(int *)((*(int *)$arg0)-4))+4)
    end
    if $argc > 1
        help cppclassname
    end
end
document cppclassname
Get's the C++ class name associated with a pointer object.
Usage: cppclassname <ptr>
end
```

About the pointer math used in cppclassname:

I'll use an example to show you how I came up with that. In the Pro Tools binary, go into sub_EB3A (the one that allocates the TPACEFailedCommand) and look for the following (there are 3 of them I believe):

v4 = TObject__operator new(72);

sub_F974(v4);

This is equivalent to the following C++ code:
*TObject *object = new TObject();*

*OK cool, that means sub_F974 is the constructor that performs basic iniitialization for the class (usually implemented in TObject::TObject()). You'll find that all over the place. So you'll notice that sub_F974 gets passed in the newly allocated memory (v4 in this case) as an argument. That's the "this" pointer in C++, and it's also the TObject *object in our pseudo-code.*
A C++ class pointer is essentially a c struct, but the first member always points to a VTable, which is an array of function pointers (the member functions).
So let's say this TObject class looks like this:

Code:

```
class TObject
{
   protected:
      int variableA;
      int variableB;
      int variableC;
      char variableD;

   public:
      TObject::TObject();
      ~TObject::TObject();

      virtual void someFunctionA();
      virtual void someFunctionB();
      virtual int doSomethingElseC();
}
```

Then that means the the pointer that gets allocated will end up looking like this:

Code:

```
object[0] = pointer to VTable
object[1] = variableA;
object[2] = variableB;
object[3] = variableC;
object[4] = variableD;
```

And the VTable that the first pointer points to will look like this:

Code:

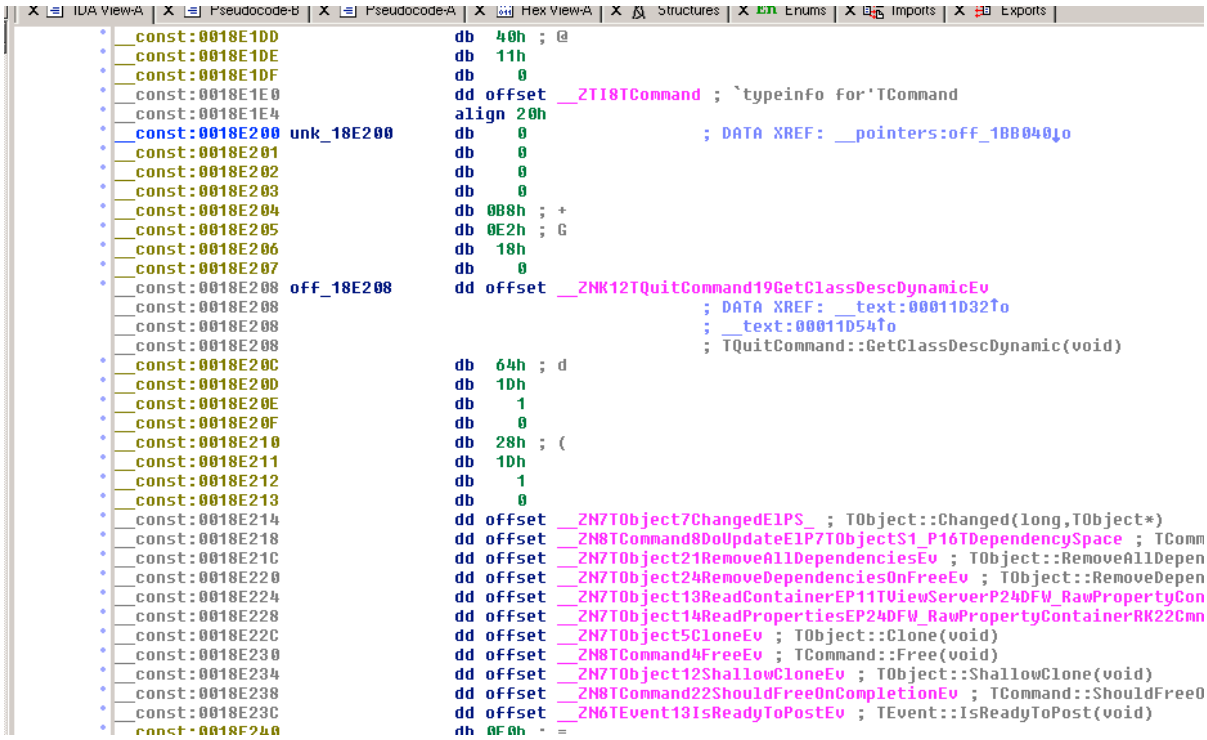
```
vtable[0] = TObject::TObject()
vtable[1] = ~TOBject::TObject()
vtable[2] = ptr to someFunction()
vtable[3] = ptr to someFunctionB()
vtable[4] = ptr to doSomethingElseC()
```

BUT, right before the structure that holds the VTable is a pointer to a typeInfo structure, which holds the class name. You can see this clearly in IDA when you go into sub_F974. In there, you'll see this:

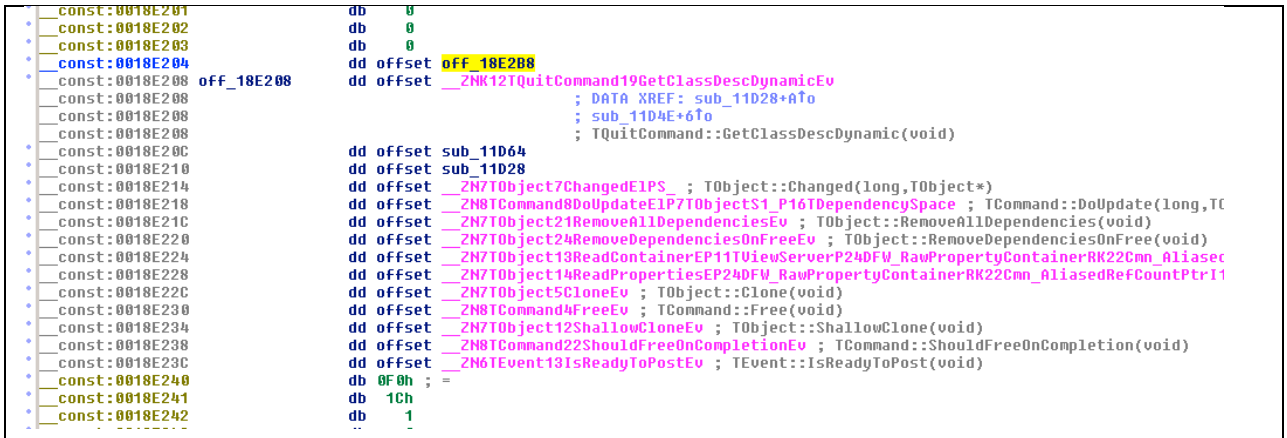
Code:

```
TQuitCommand__TQuitCommand(a1);
*(_DWORD *)a1 = (char *)&unk_18E200 + 8;
return TQuitCommand__IQuitCommand(a1, 36);
```

Here, object[0] is getting assigned unk_18E200+8, so unk_18E208 is the VTable. Double-click it and follow it ahead 8 bytes in IDA Pro. You'll see this:



Here you see that once you add 8 to unk_18E200 you end up at off_18E208. This is the VTable, and you can see an array of functions there. But you also see some random bytes... just fix these yourself by clicking on address 18E204 and pressing "d" on your keyboard twice. Do the same for 18E20C and 18E210, and this is what you'll end up with:


__const:0018E201 db 0
__const:0018E202 db 0
__const:0018E203 db 0
__const:0018E204 dd offset off_18E2B8
__const:0018E208 off_18E208 dd offset __ZNK12TQuitCommand19GetClassDescDynamicEv
; DATA XREF: sub_11D28+A↑o
; sub_11D4E+6↑o
; TQuitCommand::GetClassDescDynamic(void)
__const:0018E20C dd offset sub_11D64
__const:0018E210 dd offset sub_11D28
__const:0018E214 dd offset __ZN7TObject7ChangedElPS_ ; TObject::Changed(long,TObject*)
__const:0018E218 dd offset __ZN8TCommand8DoUpdateElP7TObjectS1_P16TDependencySpace ; TCommand::DoUpdate(long,TC
__const:0018E21C dd offset __ZN7TObject21RemoveAllDependenciesEv ; TObject::RemoveAllDependencies(void)
__const:0018E220 dd offset __ZN7TObject24RemoveDependenciesOnFreeEv ; TObject::RemoveDependenciesOnFree(void)
__const:0018E224 dd offset __ZN7TObject13ReadContainerEP11TViewServerP24DFW_RawPropertyContainerRK22Cmn_Aliasec
__const:0018E228 dd offset __ZN7TObject14ReadPropertiesEP24DFW_RawPropertyContainerRK22Cmn_AliasedRefCountPtrI1
__const:0018E22C dd offset __ZN7TObject5CloneEv ; TObject::Clone(void)
__const:0018E230 dd offset __ZN8TCommand4FreeEv ; TCommand::Free(void)
__const:0018E234 dd offset __ZN7TObject12ShallowCloneEv ; TObject::ShallowClone(void)
__const:0018E238 dd offset __ZN8TCommand22ShouldFreeOnCompletionEv ; TCommand::ShouldFreeOnCompletion(void)
__const:0018E23C dd offset __ZN6TEvent13IsReadyToPostEv ; TEvent::IsReadyToPost(void)
__const:0018E240 db 0F0h ; =
__const:0018E241 db 1Ch
__const:0018E242 db 1

So here, off_18E2B8 is the address to the typeinfo structure. So double-click off_18E2B8, and it'll take you there in IDA. Once you get there, highlight address 0x18E2BC and press "d" three times to fix it. And there you'll see the string "TAppQuitCommand". So now we know the class name, what functions it has (it's VTable), and the variables it uses.

*Now, to explain the pointer math, now that you have this background... What it's doing is dereferencing the pointer you pass it by using *(int *). This gives you the address to the VTable. Then it's subtracting 4 from that address and dereferencing it to give you the address of the typeinfo structure, and then it's adding 4 to that address to give you the address to the string... then finally using gdb's x/s to print it out.*

As you can see, you type "cppclassname 0x12345" to get the class name associated with the pointer. So re-source your gdbinit file once you've saved it with this command. Now, do cppclassname on the pointer at [ebp+8], and you get 18TPACEFailedCommand!!!
So it looks like the command we're processing is a TPACEFailedCommand! Well that's good to know, means we've found the right object! Now for the fun part. Perform the following command to get the targets (Pro Tools) pid...

Code:

```
i pid
```

This should give you something like 2317. Now we have the process ID for Pro Tools. There's a nifty little command line tool called malloc_history that will search the malloc logs that we've enabled with MallocStackLogging and tell us the history of allocations for a pointer... We can run it from gdb with the "shell" command:

Code:

```
shell malloc_history
malloc_history <pid> <address>
      Provides the allocation history all malloc blocks containing <address>
```

```
for process <pid>
     Must have enabled recording by 'setenv MallocStackLoggingNoCompact 1'
    or:
    malloc_history <pid> -all_by_size | -all_by_count | -all_events
     Must have enabled recording by 'setenv MallocStackLogging 1'
```

So we can see that we run it with the pid and then the pointer that we want to examine.

Code:

```
shell malloc_history 2317 0x12345
```

Now, this gives you a back trace of the calls that led up to that object being allocated. We're gonna examine the most recent one (the bottom one), but sometimes you may have to look at more than that. Here's what I got:

Code:

```
ALLOC 0x2db0ace0-0x2db0ad27 [size=72]: thread_a0179500
0xe8cd
0xe99f
0xf8c2
0xebcf
TObject::operator new(unsigned long)
operator new(unsigned long, std::nothrow_t const&)
malloc
malloc_zone_malloc
```

Of course I made it look pretty in TextEdit so it was easier to read. But here you can see that the TObject::operator new was called, and we see the address that called it was 0xEBCF!! So that's the location where the TPACEFailedCommand was allocated, and hence where some check failed! We can also see the addresses that led up to that. Examine all of those places in IDA Pro until you can see what's happening.

In this case, I backed up to 0xF8C2, and looked at the call immediately before it, sub_EB3A. It may even be easier to start from the beginning and follow the calls forward .... sub_EB3A is where the TPACEFailedCommand is allocated from, so what leads to sub_EB3A being called?

```
sub_11882(&v22);
v18 = v23;
v19 = v24;
v20 = v25;
if ( (unsigned __int8)sub_105D0(v22) )
{
  v14 = operator new(8, v18, v19, v20);
  Sys_DLL__Sys_DLL(v14);
  Sys_FileLoc__Sys_FileLoc(&v30);
  Sys_FileLocUtils__GetApplicationLoc(&v29);
  Sys_FileLoc__operator_(&v30);
  Sys_FileLoc___Sys_FileLoc(&v29);
  Sys_FileLocUtils__GetApplicationBundleFrameworksLoc(&v30);
  Sys_FileType__Sys_FileType(&v28, &Sys_FileType__cFolder);
  Sys_FileLoc__Sys_FileLoc(&v27, &v30, &dword_1B8F84, &v28);
  Sys_FileLoc__operator_(&v30);
  Sys_FileLoc___Sys_FileLoc(&v27);
  Sys_FileType___Sys_FileType(&v28, &v27);
  Sys_File__Sys_File(&v26, &v30);
  v15 = Sys_File__IsOnline(&v26);
  Sys_File___Sys_File(&v26);
  if ( v15 )
  {
    (*(void (__cdecl **)(int, char *, signed int))(*(_DWORD *)v14 + 8))(v14, &v30, 1);
    std__string__string(&v31, "LaunchProTools", &v37);
    v16 = (*(int (__cdecl **)(int, int *))(*(_DWORD *)v14 + 20))(v14, &v31);
    std__string___Rep___M_dispose(v31 - 12);
    if ( v16 )
    {
      ((void (__cdecl *)(char *, char *))v16)(&v21, &v36);
    }
    else
    {
      if ( dword_1AA484 )
      {
        if ( (unsigned __int8)unk_1AA480(134221327, &v36) )
          unk_1AA490(134221327, "LoadAndRunProToolsDLL - unable to find launch entry point in ProTools dll.");
      }
    }
  }
  Sys_FileLoc___Sys_FileLoc(&v30);
}
else
{
  sub_EB3A();
}
```

Ahhh.. Immediately after the checkIlokAuth function, sub_105D0 gets called. If that returns a 0, sub_EB3A get's called, and then the program exits... But if it returns 1, then it goes on to run the program! You all know what to do here: patch sub_105D0 to always return 1.

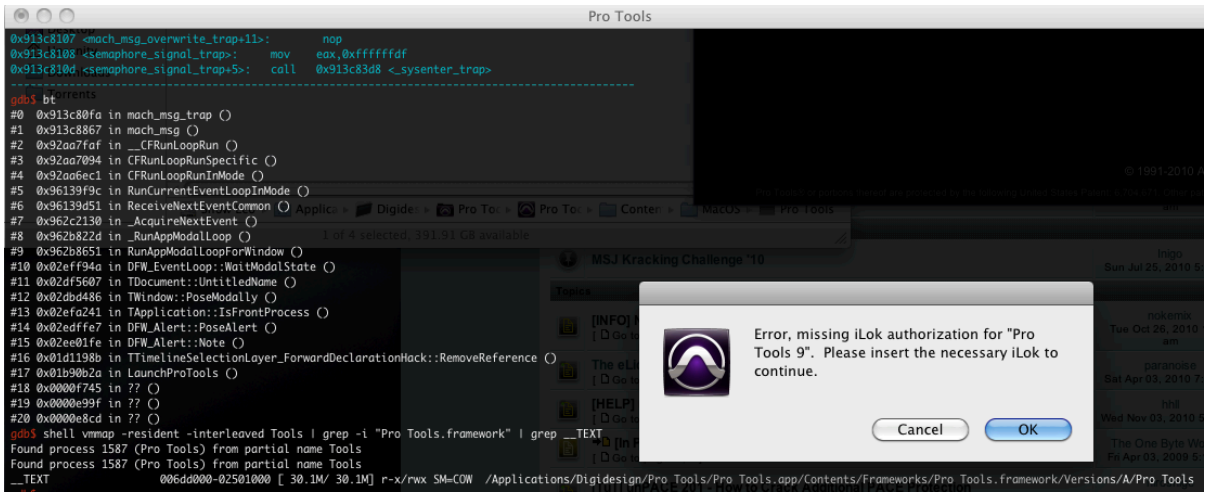
Please, do this step a couple times, look and make sure you understand how I arrived at the conclusion I did. This understanding will open up many more doors for you.

Step 3: Deja Vu

or, ***Haven't I Seen This Dialog Somewhere Before?***

OK, so now we've patched those 2 places in the binary, the call that checks the ilok for a valid license, and the call that checks the components. Some of you may be wondering why we're patching at all, given what I've said about the checksumming. Well, I didn't know that about the prot first, had to make some patches to figure it out, and I wanna show you how I figured the checksumming out, not just ask you to take my word for it. And anyway, we'll still need to know the vulnerable places

So with those 2 places patched, we run Pro Tools under gdb, and we get the same dialog again!


Pro Tools
0x913c8107 <mach_msg_overwrite_trap+11>: nop
0x913c8108 <semaphore_signal_trap>: mov eax,0xfffffffd
0x913c810d <semaphore_signal_trap+5>: call 0x913c83d8 <_sysenter_trap>
gdb$ bt
#0 0x913c80fa in mach_msg_trap ()
#1 0x913c8867 in mach_msg ()
#2 0x92aa7faf in __CFRunLoopRun ()
#3 0x92aa7094 in CFRunLoopRunSpecific ()
#4 0x92aa6ec1 in CFRunLoopRunInMode ()
#5 0x96139f9c in RunCurrentEventLoopInMode ()
#6 0x96139d51 in ReceiveNextEventCommon ()
#7 0x962c2130 in _AcquireNextEvent ()
#8 0x962b822d in _RunAppModalLoop ()
#9 0x962b8651 in RunAppModalLoopForWindow ()
#10 0x02eff94a in DFW_EventLoop::WaitModalState ()
#11 0x02df5607 in TDocument::UntitledName ()
#12 0x02dbd486 in TWindow::PoseModally ()
#13 0x02efa241 in TApplication::IsFrontProcess ()
#14 0x02edffe7 in DFW_Alert::PoseAlert ()
#15 0x02ee01fe in DFW_Alert::Note ()
#16 0x01d1198b in TTimelineSelectionLayer_ForwardDeclarationHack::RemoveReference ()
#17 0x01b90b2a in LaunchProTools ()
#18 0x0000f745 in ?? ()
#19 0x0000e99f in ?? ()
#20 0x0000e8cd in ?? ()
gdb$ shell vmmap -resident -interleaved Tools | grep -i "Pro Tools.framework" | grep __TEXT
Found process 1587 (Pro Tools) from partial name Tools
Found process 1587 (Pro Tools) from partial name Tools
__TEXT 006dd000-02501000 [30.1M/ 30.1M] r-x/rwx SM=COW /Applications/Digidesign/Pro Tools/Pro Tools.app/Contents/Frameworks/Pro Tools.framework/Versions/A/Pro Tools
Error, missing iLok authorization for "Pro Tools 9". Please insert the necessary iLok to continue.
Cancel
OK

Here you'll notice I did the same thing... ctrl+c to break in gdb, issued a backtrace... But this time I noticed the last address was after LaunchProTools, and it was a higher address... 0x01D1198B. So as you can see, I ran vmmap and looked for the __TEXT section with the address range that the address fit into, and you can see it's in the Pro Tools.framework. We are interested in the address 0x01D1198B because it's the one right before DRW_Alert::Note(), and we know that that's the one that brings on the dialog box. So pop open IDA Pro and load the Pro Tools Framework binary... I have to warn you, this one will take about 30 minutes for IDA to analyze, so go grab some coffee whle you wait...

Now, to find the offset we need to look at in IDA Pro, we have to take the address and subtract the base. Looking at vmmap, we see the __TEXT section for the Pro Tools framework is at 0x6DD000 - 0x2501000 for me (it may be different for you, or it might not.. I actually think it will be the same).
__TEXT 006dd000-02501000 [30.1M/ 30.1M] r-x/rwx SM=COW
So we need to subtract 0x6DD000 from the address in the backtrace, 0x01D1198B, which give us 0x163498B. So in IDA Pro, go to your IDA-View A, then select "Jump" from the menu, and then "Jump to Address"... and type in 0x163498B. Enter decompilation mode, and this is what we get:

```
char v74; // [sp+9Ch] [bp-1Ch]@3
char v75; // [sp+9Fh] [bp-19h]@3
signed int v76; // [sp+BCh] [bp+4h]@45

while ( g_IsAuthorized )
{
  g_IsAuthorized = checkIlokAuth(0) == 0;
  if ( !g_IsAuthorized )
    break;
  sub_2103AA(&v65);
  sub_210376(&unauthorizedMessage);
  alertResult = DFW_Alert__Note(&unauthorizedMessage, DFW_Alert__sOKCancel, 0, 0, 1, -1, 0, 0);
  std__string___Rep___M_dispose(unauthorizedMessage - 12, &v75);
  std__string___Rep___M_dispose(v65 - 12, &v74);
  if ( alertResult )
    cancelAllTasks();
}
sub_163323E();
HIDWORD(v71) = (char *)loc_4474D8 + 353;
v1 = (int)((char *)loc_4B2D2C + 350);
LODWORD(v71) = (char *)loc_4B2D2C + 350;
if ( !dword_1FA2CAC )
  SPSStartupOptionSInt32  operator long(v1);
```

(Notice that I went ahead and gave meaningful names here...) Be sure to poke around this place until you're familiar with it, go ahead and explore a little bit. This chunk of code that spawns the "no ilok" message seems incredibly familiar to the one we patched before... Looks like we need to patch this checkIlokAuth to also return 1
Now, as you explore this function, and other function in this binary, you'll see things like this:


```
v3 -= 78690203;
v4 = __ROR__(1783543879, 25);
v5 = __ROR__(
       v3 ^ (((_BYTE)v3 ^ (unsigned __int8)(v3 >> 5)) & 7 | 32 * (((_BYTE)v3 ^ (unsigned __int8)(v3 >> 5)) & 7)),
       29);
v6 = __ROR__(v5 - 58000637, 12);
v7 = ~(v6 - 732251818);
v8 = (((unsigned int)v7 >> 18) ^ (unsigned __int16)((unsigned int)v7 >> 2)) & 0x3FFF;
v9 = v7 ^ ((v8 << 18) | 4 * v8);
v10 = __ROR__(v7 ^ ((v8 << 18) | 4 * v8), 28);
v9 = __ROR__(v9, 4);
v11 = v9 & 0x96969696 | v10 & 0x69696969;
v12 = __ROR__(
        ~(((v4 + 179689957) ^ (((((unsigned __int16)((unsigned int)(v4 + 179689957) >> 16) ^ (unsigned __int16)((unsigned int)(v4 + 179689957) >> 5)) & 0x3
          + 366672377) ^ (((unsigned __int8)~(((v4 - 27) ^ 32
                                                * (((unsigned int)(v4 + 179689957) >> 16) ^ ((unsigned int)(v4 + 179689957) >> 5)) ^ 0xEA)
                                          - 7) ^ (unsigned __int8)(~(((v4 + 179689957) ^ (((((unsigned __int16)((unsigned int)(v4 + 179689957) >> 16) ^ (un
                                                                + 366672377) >> 7)) & 0x3F | ((((unsigned __int8)~(((v4 - 27) ^ 32 * (((unsigned int)(v4
        16);
v13 = v11 ^ v12;
v14 = v11 ^ v13 & 0x33333333;
v15 = v11 ^ v13 & 0xCCCCCCCC;
v16 = __ROR__(v14, 24);
v1 = __ROR__(v14, 8);
v17 = (v1 & 0x2DD22DD2 | v16 & 0xD22DD22D) + 1713450793;
v18 = __ROR__(v15, 24);
v15 = __ROR__(v15, 8);
v69 = 0;
v68 = 0;
v62 = ((int (__cdecl *)(int, int))sub_162F226)(v17, (v15 & 0xC837C837 | v18 & 0x37C837C8) + 1091516016);
v63 = v19;
```

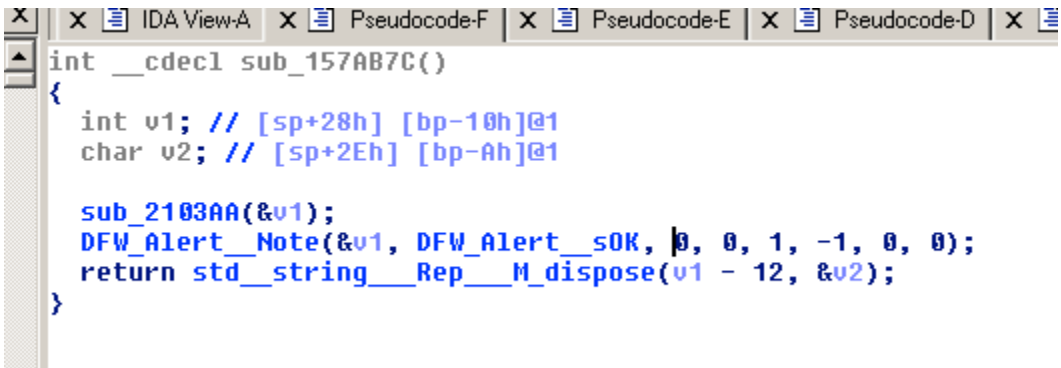
All those ROR, xor's (^), and the crazy numbers that don't seem to make sense, you'll notice they're in every single function... these are the inline checksum algo's. As you study them, you'll begin to notice patterns and eventually you'll understand what they're doing. Right now, they're not our focus, but just be aware of them, because as you study the code, it's your goal to understand as much of it as you can.

OK, once we've made the patch to the checkIlokAuth function in the Pro Tools.framework, let's go ahead an load it up in gdb again... This time, we get pretty far, all the way past the splash screen (you may have to load an antiptrace kext here, or break on ptrace and return, because it loads PACE plugins that call PT_DENY_ATTACH and will kick you out of gdb). But finally we're greeted with another dialog just like the one before, the "unrecognized component" dialog. Treat this one exactly the same as you did the other one... except if you look at the backtrace, you'll see this:

Code:

```
#13 0x02efa241 in TApplication::IsFrontProcess ()
#14 0x02edffe7 in DFW_Alert::PoseAlert ()
#15 0x02ee01fe in DFW_Alert::Note ()
#16 0x01c57be2 in
TTimelineSelectionLayer_ForwardDeclarationHack::RemoveReference ()
#17 0x02def2ed in TCommandHandler::PerformCommand ()
#18 0x02def8b3 in TCommand::Process ()
#19 0x02de77c6 in TApplication::ProcessEvent ()
#20 0x02de7a20 in TApplication::WakeUp ()
#21 0x02ef9077 in TApplication::GetApplicationVersionComponents ()
#22 0x9610cf2f in DispatchEventToHandlers ()
#23 0x9610c1f6 in SendEventToEventTargetInternal ()
#24 0x9610c055 in SendEventToEventTargetWithOptions ()
#25 0x96140bb0 in ToolboxEventDispatcherHandler ()
#26 0x9610d380 in DispatchEventToHandlers ()
#27 0x9610c1f6 in SendEventToEventTargetInternal ()
#28 0x9612e9bb in SendEventToEventTarget ()
#29 0x962b7cc3 in ToolboxEventDispatcher ()
#30 0x962b7dfb in RunApplicationEventLoop ()
#31 0x02eff749 in DFW_EventLoop::RunApplicationEventLoop ()
#32 0x02de73e7 in TApplication::Run ()
#33 0x01b9175e in LaunchProTools ()
#34 0x0000f745 in ?? ()
#35 0x0000e99f in ?? ()
#36 0x0000e8cd in ?? ()
```

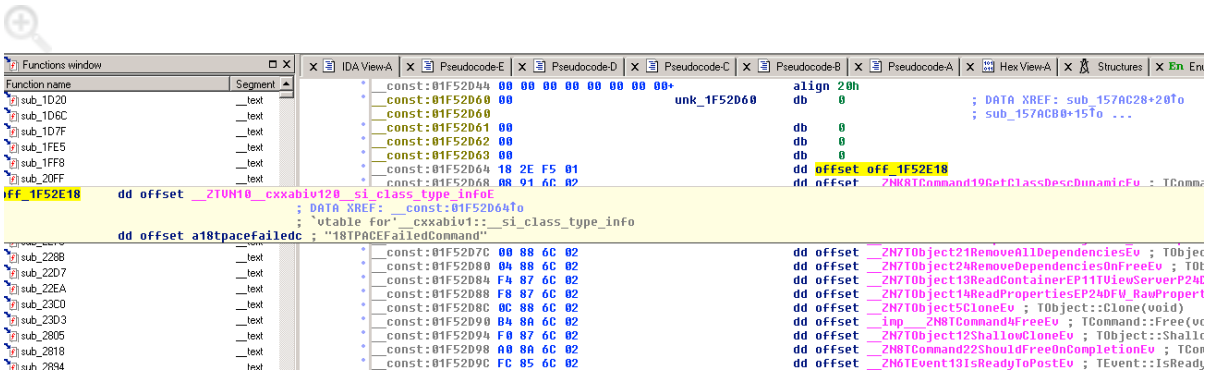
And you'll notice there's a DFW_Alert::Note() in there! And the address right before that is 0x01C57BE2... so subtract the base and go to there in IDA Pro (0x157ABE2)... It takes us here:



which is about as useful as a broken toaster. So go back out of decompilation mode to the disassembly view (the one without the graphs, if you have graphs, press spacebar to toggle plain disassembly view). Scroll up to the beginning of the function (we want to check the cross-references... see what calls this piece of code). This is what the top of the function looks like:

```
__text:0157AB7C                         ; =============== S U B R O U T I N E =======================================
__text:0157AB7C
__text:0157AB7C                         ; Attributes: bp-based frame
__text:0157AB7C
__text:0157AB7C                         sub_157AB7C     proc near               ; DATA XREF: __const:01F52DC0↓o
__text:0157AB7C
__text:0157AB7C                         var_10          = dword ptr -10h
__text:0157AB7C                         var_A           = byte ptr -0Ah
__text:0157AB7C                         var_9           = byte ptr -9
__text:0157AB7C
__text:0157AB7C 55                                      push    ebp
__text:0157AB7D 89 E5                                   mov     ebp, esp
__text:0157AB7F 56                                      push    esi
__text:0157AB80 53                                      push    ebx
__text:0157AB81 83 EC 30                                sub     esp, 30h
__text:0157AB84 E8 00 00 00 00                          call    $+5
__text:0157AB89 5B                                      pop     ebx
__text:0157AB8A 8D 75 F0                                lea     esi, [ebp+var_10]
__text:0157AB8D C7 44 24 04 36 02 00 00                 mov     dword ptr [esp+4], 236h
__text:0157AB95 89 34 24                                mov     [esp], esi
__text:0157AB98 E8 0D 58 C9 FE                          call    sub_2103AA
__text:0157AB9D 83 EC 04                                sub     esp, 4
__text:0157ABA0 C7 44 24 1C 00 00 00 00                 mov     dword ptr [esp+1Ch], 0
__text:0157ABA8 C7 44 24 18 00 00 00 00                 mov     dword ptr [esp+18h], 0
__text:0157ABB0 C7 44 24 14 FF FF FF FF                 mov     dword ptr [esp+14h], 0FFFFFFFFh
__text:0157ABB8 C7 44 24 10 01 00 00 00                 mov     dword ptr [esp+10h], 1
__text:0157ABC0 C7 44 24 0C 00 00 00 00                 mov     dword ptr [esp+0Ch], 0
__text:0157ABC8 C7 44 24 08 00 00 00 00                 mov     dword ptr [esp+8], 0
__text:0157ABD0 8B 83 77 E9 8A 00                       mov     eax, ds:(__ZN9DFW_Alert3sOKE_ptr - 157AB89h)[ebx]
__text:0157ABD6 89 44 24 04                             mov     [esp+4], eax
__text:0157ABDA 89 34 24                                mov     [esp], esi
__text:0157ABDD E8 E4 88 23 00                          call    __ZN9DFW_Alert4NoteERKSsRSt6vectorISsSaISsEEPS4_lllPbPS0_ ; DFW_Alert::Note(std::str
```

Here, we'll notice the DATA XREF. Right-click on it and go to "jump to cross reference..." and click it. There's only one there so go to it. It takes us to what looks like a VTable! So that means that there's some class that calls this function that displays the dialog, and the function is basically one of the class methods... but which class? Scroll up to the where the VTable begins, and you'll see the top one says "offset off_1F52E18". So mouse over that and use the scroll wheel to unveil more of it, and we see that i'ts the TPACEFailedCommand again!


Functions window
Function name
Segment
sub_1D20
sub_1D6C
sub_1D7F
sub_1FE5
sub_1FF8
sub_20FF
sub_228B
sub_22D7
sub_22EA
sub_23C0
sub_23D3
sub_2805
sub_2818
IDA View-A
Pseudocode-E
Pseudocode-D
Pseudocode-C
Pseudocode-B
Pseudocode-A
Hex View-A
Structures
__const:01F52D44 00 00 00 00 00 00 00 00+ align 20h
__const:01F52D60 00 unk_1F52D60 db 0 ; DATA XREF: sub_157AC28+20↑o
__const:01F52D60 ; sub_157AC80+15↑o ...
__const:01F52D61 00 db 0
__const:01F52D62 00 db 0
__const:01F52D63 00 db 0
__const:01F52D64 18 2E F5 01 dd offset off_1F52E18
off_1F52E18 dd offset __ZTVN10__cxxabiv120__si_class_type_infoE
; DATA XREF: __const:01F52D64↑o
; `vtable for'__cxxabiv1::__si_class_type_info
dd offset a18tpacefailedc ; "18TPACEFailedCommand"
__const:01F52D7C 00 88 6C 02 dd offset __ZN7TObject21RemoveAllDependenciesEv ; TObjec
__const:01F52D80 04 88 6C 02 dd offset __ZN7TObject24RemoveDependenciesOnFreeEv ; TOb
__const:01F52D84 F4 87 6C 02 dd offset __ZN7TObject13ReadContainerEP11TViewServerP24
__const:01F52D88 F8 87 6C 02 dd offset __ZN7TObject14ReadPropertiesEP24DFW_RawPropert
__const:01F52D8C 0C 88 6C 02 dd offset __ZN7TObject5CloneEv ; TObject::Clone(void)
__const:01F52D90 B4 8A 6C 02 dd offset __imp___ZN8TCommand4FreeEv ; TCommand::Free(vo
__const:01F52D94 F0 87 6C 02 dd offset __ZN7TObject12ShallowCloneEv ; TObject::Shallo
__const:01F52D98 A0 8A 6C 02 dd offset __ZN8TCommand22ShouldFreeOnCompletionEv ; TCom
__const:01F52D9C FC 85 6C 02 dd offset __ZN6TEvent13IsReadyToPostEv ; TEvent::IsReady

Hopefully you guys are beginning to understand how important it is to understand the structure of a C++ object and why I'm taking so much time to explain it... It come in incredibly handy. Watch...

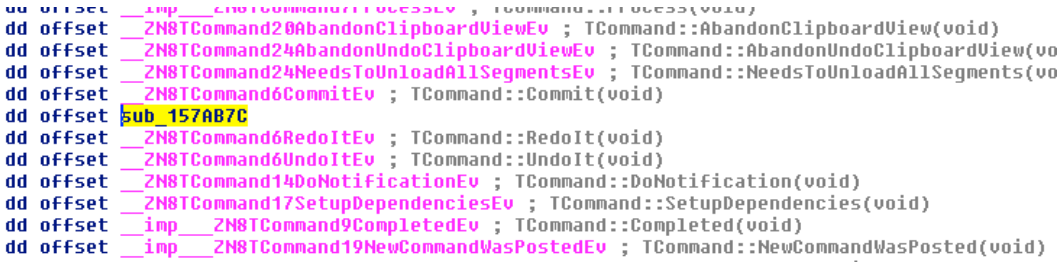
So from here, what I would do is set restart gdb and set a breakpoint on TCommand::Process() again, and see what commands come through by using cppclassname on the first argument ([ebp+8]) of that

function, until we find the TPACEFailedCommand, but just to also get a feel for what's happening in this mysterious event loop

So the first break on TCommand::Process shows us a LaunchProTools command... cool. The second break (well into the splash screen, seems the LaunchProTools command covers quite a bit of territory) shows us a TPACECheckCommand... what??? Interesting... the command after that one is the TPACEFailedCommand. Ahh, so we should have a good idea of what's going on now.

Pro Tools is launched by sending the LaunchProTools command to the event loop to be processed. This command does all the initialization work. After that command is sent, it sends another command, the TPACECheckCommand, which checks the components to make sure the PACE protection is still intact and unmodified (basically, to check that the frameworks and such are still PACE'd). If it fails, a TPACEFailedCommand is sent. So it would seem that the patch is probably going to be in the TPACECheckCommand, because that's where it seems like the check is made. But how do we find that command in the code?

Two ways, depending on how you want to go about it (I suggest if you've never solved a problem this way that you go through both before deciding which one you're going to use). First way, you can break on the TCommand::Process and trace your way into the calls until you find where the call to process the command is made, all the while following along in IDA Pro in decompilation mode so you can get a higher level understanding of what's going on. But I'm going to show you the second way. This way takes a bit of educated guessing but will almost always be faster. First, find the VTable for the TPACEFailedCommand again by backtracing from the DFW_Alert::Note(). It's at offset 0x1F52D64 in the framework. Now scroll down... After that VTable structure, you'll see another one begin at offset 0x1F52E44. Mouse over the first entry (offset off_1F52EF8) to get the class name... Hmmm looks like a TPACEExitCommand... cool we're finding out more commands! Now keep going to the next one, it's at offset 0x1F52F24 and mouse over the first entry (offset off_1F52FD8). Hmmm, it's the TPACECheckCommand! Now if you study the VTable for the TPACEFailedCommand, you'll see that the call that we backtraced from (the one that TCommand::Process actually calls) was sub_157AB7C...


dd offset __ZN8TCommand20AbandonClipboardViewEv ; TCommand::AbandonClipboardView(void)
dd offset __ZN8TCommand24AbandonUndoClipboardViewEv ; TCommand::AbandonUndoClipboardView(vo
dd offset __ZN8TCommand24NeedsToUnloadAllSegmentsEv ; TCommand::NeedsToUnloadAllSegments(vo
dd offset __ZN8TCommand6CommitEv ; TCommand::Commit(void)
dd offset sub_157AB7C
dd offset __ZN8TCommand6RedoItEv ; TCommand::RedoIt(void)
dd offset __ZN8TCommand6UndoItEv ; TCommand::UndoIt(void)
dd offset __ZN8TCommand14DoNotificationEv ; TCommand::DoNotification(void)
dd offset __ZN8TCommand17SetupDependenciesEv ; TCommand::SetupDependencies(void)
dd offset __imp__ZN8TCommand9CompletedEv ; TCommand::Completed(void)
dd offset __imp__ZN8TCommand19NewCommandWasPostedEv ; TCommand::NewCommandWasPosted(void)

You'll notice that it's right after TCommand::Commit() and before TCommand::RedoIt() and has no

name... So lets' look for the same thing in TPACECheckCommand...

```
dd offset __ZN8TCommand11IsRecurringEv ; TCommand::IsRecurring(void)
dd offset __ZN6TEvent10GetHandlerEv ; TEvent::GetHandler(void)
dd offset __imp___ZN8TCommand7ProcessEv ; TCommand::Process(void)
dd offset __ZN8TCommand20AbandonClipboardViewEv ; TCommand::AbandonClipboardView(void)
dd offset __ZN8TCommand24AbandonUndoClipboardViewEv ; TCommand::AbandonUndoClipboardView(voi
dd offset __ZN8TCommand24NeedsToUnloadAllSegmentsEv ; TCommand::NeedsToUnloadAllSegments(voi
dd offset __ZN8TCommand6CommitEv ; TCommand::Commit(void)
dd offset sub_157B4F8
dd offset __ZN8TCommand6RedoItEv ; TCommand::RedoIt(void)
dd offset __ZN8TCommand6UndoItEv ; TCommand::UndoIt(void)
dd offset __ZN8TCommand14DoNotificationEv ; TCommand::DoNotification(void)
dd offset __ZN8TCommand17SetupDependenciesEv ; TCommand::SetupDependencies(void)
dd offset __imp___ZN8TCommand9CompletedEv ; TCommand::Completed(void)
dd offset __imp___ZN8TCommand19NewCommandWasPostedEv ; TCommand::NewCommandWasPosted(void)
dd offset __ZN8TCommand15NoLongerTopUndoEv ; TCommand::NoLongerTopUndo(void)
```

And we find it right there, sub_157B4F8. Basically we can deduce that these function right there are the "meat" of the command... it's the function that all subclasses of TCommand override to perform their specific role. (You gotta think like you own the source code). BTW, you could have also used the "Search Text..." function in IDA Pro to search for all occurrences of "TPACECheckCommand" and found it that way 😉But now that we've found the function that performs the pace check (sub_157B4F8), let's go into it and enter decompilation mode.

Sigh. Here we can see that about the first 75% of the function is one big checksum algorithm. Tell me that this doesn't slow down the program... Anyways, at the bottom of the function you'll see this:

```
}
if ( (v70 ^ (v70 >> 10) ^ ((v70 ^ (v70 >> 10)) << 20)) == -192078635 )
{
  v67 = *(_DWORD *)(a1 + 76);
  v68 = *(_DWORD *)(a1 + 80);
  v69 = *(_DWORD *)(a1 + 84);
  result = (char *)sub_157E528(*(_DWORD *)(a1 + 72));
  if ( !(_BYTE)result )
  {
    if ( *(_DWORD *)off_1E29490 )
    {
      v67 = 0;
      sub_106A19A(*(_DWORD *)off_1E29490);
    }
    v64 = gApplication_ptr;
    v63 = *(_DWORD *)(*(_DWORD *)gApplication_ptr + 44);
    if ( v63 )
      (*(void (__cdecl **)(_DWORD, int, int, int))(*(_DWORD *)v63 + 68))(
        *(_DWORD *)(*(_DWORD *)gApplication_ptr + 44),
        v67,
        v68,
        v69);
    v65 = TObject__operator new(72);
    sub_157ACA6(v65);
    (*(void (__cdecl **)(_DWORD, int, int, int))(**(_DWORD **)v64 + 216))(*(_DWORD *)v64, v65, v68, v69);
    v66 = TObject__operator new(72);
    sub_157ADF2(v66);
    result = (char *)(*(int (__cdecl **)(_DWORD, int))(**(_DWORD **)v64 + 216))(*(_DWORD *)v64, v66);
  }
```

It does a few checks, and then we see the TObject_operator new... which means it's allocating some objects. It's really easy to tell what objects it's allocating from right here in IDA Pro. See where it says v65

= TObject_operator new in my output... Here's how we'll tell what that object is Immediately after v65 get's initialized from the new command, it's passed into sub_157ACA6 as the only argument. Follow that call in. Inside this call, it basically just calls another method with the same pointer as an argument (sub_157AC28). Follow that call. In this call we see

Code:

```
TCommand__TCommand(a1);
  *(_DWORD *)a1 = (char *)&unk_1F52D60 + 8;
  return TCommand__ICommand(a1, 0, *(_DWORD *)gApplication_ptr, 0, 0, 0);
```

So here the first entry in a1 (which is our pointer from the new command) is getting assigned unk_1F52D60+8... and we know the first entry is our VTable, right? So follow unk_1F52D0 and then go 8 bytes further and look at the typeinfo pointer... Looks like this is the TPACEFailedCommand. So we know what that v65 is now, so let's go back and rename stuff so we know what's going on.

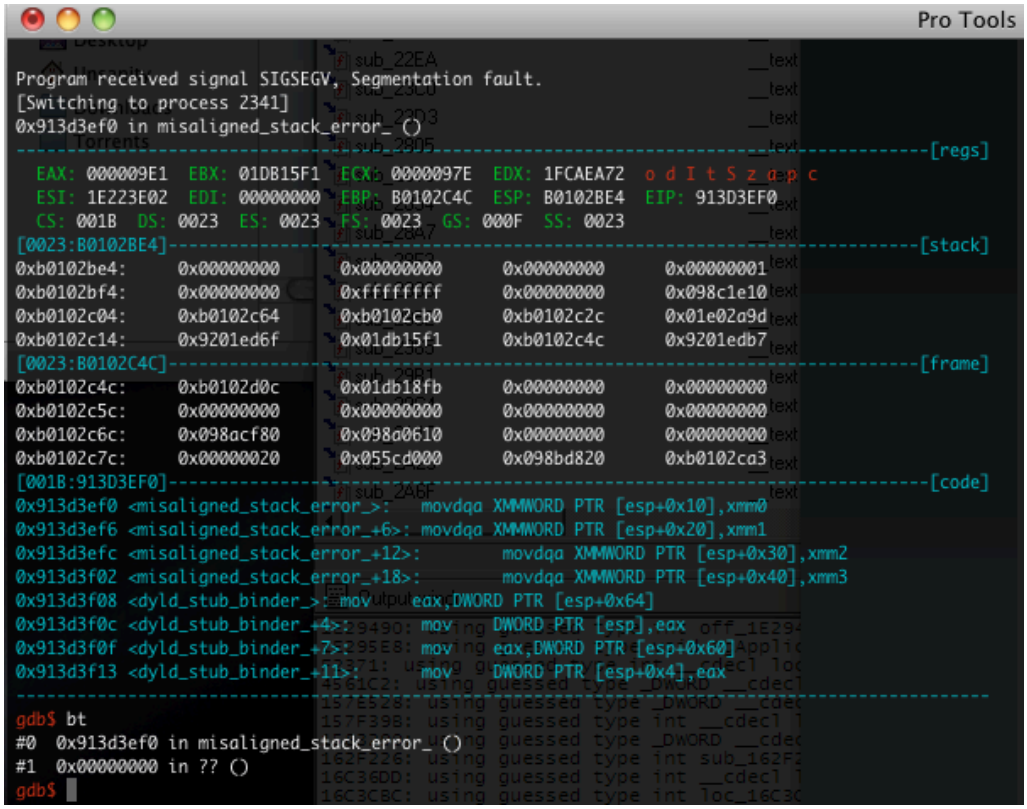
```
v69 = *(_DWORD *)(a1 + 84);
result = (char *)checkPaceComponents(*(_DWORD *)(a1 + 72));
if ( !(_BYTE)result )
{
  if ( *(_DWORD *)off_1E29490 )
  {
    v67 = 0;
    sub_106A19A(*(_DWORD *)off_1E29490);
  }
  v64 = gApplication_ptr;
  v63 = *(_DWORD *)(*(_DWORD *)gApplication_ptr + 44);
  if ( v63 )
    (*(void (__cdecl **)(_DWORD, int, int, int))(*(_DWORD *)v63 + 68))(
      *(_DWORD *)(*(_DWORD *)gApplication_ptr + 44),
      v67,
      v68,
      v69);
  paceFailedCommand = TObject__operator new(72);
  sub_157ACA6(paceFailedCommand);
  (*(void (__cdecl **)(_DWORD, int, int, int))(**(_DWORD **)v64 + 216))(*(_DWORD *)v64, paceFailedCommand, v68, v69);
  paceExitCommand = TObject__operator new(72);
  sub_157ADF2(paceExitCommand);
  result = (char *)(*(int (__cdecl **)(_DWORD, int))(**(_DWORD **)v64 + 216))(*(_DWORD *)v64, paceExitCommand);
}
```

So here I've renamed sub_157E528 to checkPaceComponents, because if that function returns 0, then we can see that the TPACEFaledCommand and the TPACEExitCommand get allocated and put in the event loop. So hopefully you all know what to do here, and that's patch the checkPaceComponents to return 1.

Step 4: Fixing the Patches

or, ***Segmentation Fault: The Misaligned Stack Error***

OK, after those patches are made, let's fire up Pro Tools under gdb again. This time we make it all the way throught the splash screen to the quickstart window, where we can open a project or start a new one! But our vindication is short-lived... soon after starting a new project, we crash to gdb in misaligned_stack_error... and the backtrace is truly helpful (not):



But alll we patched were some calls to return 1? It's not like we replaced whole sections of code... The first thing one might think is that some important stuff happened in the calls we bypassed, like the initialization of some global object, but some googling of misaligned stack error shows no conclusive results. One quote from someone says:

Quote:

RE: Misaligned Stack Error

The stack always has to be aligned to 16 bytes on Mac OS X/i386. This can go wrong when you use assembler code that was written for other platforms, if a wrong calling convention is specified for an external routine, or if an external routine has a different actual signature than the one specified in the source code.

Note that the mysql client library interface changes very often. So you have to distribute the mysql client

library that you use yourself with your program, since otherwise your user may have a different version and things may go horribly wrong.
-- Jonas Maebe, February 18, 2009 01:39 PM

OK, with that, it's still possible that we skipped over some crazy voodoo in the functions we bypassed. I spent about 30 minutes tracing these in IDA Pro to figure it out before I started to realize what those checksum algorithms IN EVERY FUNCTION actually were. Let's take a look at one. It's sub_157A5D6 in Pro Tools Framework. All the way at the bottom (right before the return) you'll see a compare that looks like this:

if (v25 != -173065414)
JUMPOUT(__CS__, 0x162F21Du);

I'm gonna go ahead and rename v32 to checksum and right-click on that negative number and select Hexadecimal from the menu. Then I'm going to follow backward up to see what checksum is getting assigned.

```
  v54 = v23;
}
v24 = 0;
checksum = 191;
while ( v24 != 165239 )
{
  v26 = __ROR__(checksum, 28);
  checksum = *(_BYTE *)(0x15A2D79 - v24++ - 1) ^ v26;
}
v27 = __ROR__(
        (((((unsigned int)(5 * v55) >> 13) ^ v55) >> 12) - 1
        12);
```

Here we see that it's taking a byte from memory location 0x15A2D79 plus some offset (in a loop). So v24 is the counter, and it gets increased once per loop (v24++), and as long as it is less than 165239 the loop continues. So this little loop can be rewritten like this in C:

Code:

```
checksum = 191;
char *baseAddress = 0x15A2D79;
for (int count = 1; count < 165239; count++)
{
   int temp = ROR(checksum, 28);
   checksum = *(baseAddress - count) ^ temp;
}
...
```

```
if ( checksum != -0xA50C4C6)
    jumpout( 0x162F21D);
```

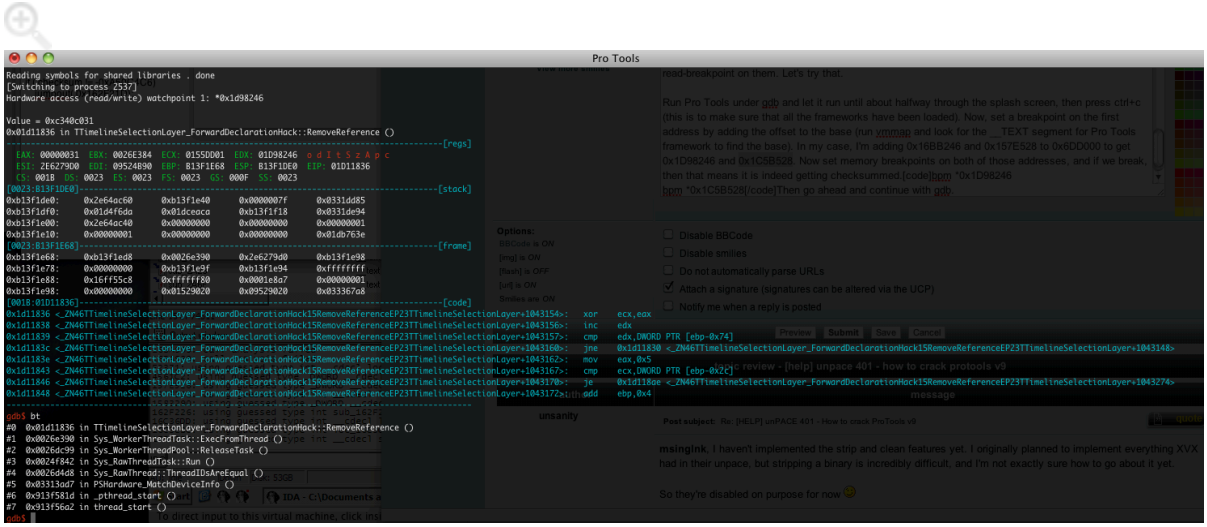
Now doesn't that address 0x15A2D79 look like an offset in the Pro Tools framework? Well it sure is. So basically what this is doing is from 0x15A2D79 and backwards 165239 bytes (so essentially from 0x157A802 - 0x15A2D79) it's checksumming and comparing it to a known good value... if it's not the correct value, it jumps to address 0x162F21D and exits the program with an error message... Hmmm, now all those random things throughout the code are starting to make sense.. it's actually chunks of the program that it's checking! So this could be causing the misaligned stack error! OK, so the addresses we modified to make the two functions in the Pro Tools framework return 1 are 0x16BB246 and 0x157E528. So we can tell if those addresses are being checksummed by setting a read-breakpoint on them. Let's try that.

Run Pro Tools under gdb and let it run until about halfway through the splash screen, then press ctrl+c (this is to make sure that all the frameworks have been loaded). Now, set a breakpoint on the addresses by adding the offset to the base (run vmmap and look for the __TEXT segment for Pro Tools framework to find the base). In my case, I'm adding 0x16BB246 and 0x157E528 to 0x6DD000 to get 0x1D98246 and 0x1C5B528. Now set memory breakpoints on both of those addresses, and if we break, then that means it is indeed getting checksummed.

Code:

```
bpm *0x1D98246
bpm *0x1C5B528
```

Then go ahead and continue with gdb. Sure enough, we break here with this backtrace:


Pro Tools
Reading symbols for shared libraries . done
[Switching to process 2537]
Hardware access (read/write) watchpoint 1: *0x1d98246
Value = 0xc340c031
0x01d11836 in TTimelineSelectionLayer_ForwardDeclarationHack::RemoveReference ()
gdb$ bt
#0 0x01d11836 in TTimelineSelectionLayer_ForwardDeclarationHack::RemoveReference ()
#1 0x0026e390 in Sys_WorkerThreadTask::ExecFromThread ()
#2 0x0026dc90 in Sys_WorkerThreadPool::ReleaseTask ()
#3 0x0024f842 in Sys_RawThreadTask::Run ()
#4 0x0026d4d8 in Sys_RawThread::ThreadIDsAreEqual ()
#5 0x03313ad7 in PSHardware_MatchDeviceInfo ()
#6 0x913f581d in _pthread_start ()
#7 0x913f56a2 in thread_start ()

Notice at the very top, when gdb breaks, it prints the value of the memory that was accessed. In this case it says "Value = 0xc340c031", which is 0x31C040C3, what I used to make the function return 1.

Additionally, from the backtrace, we can tell that this check is running from a thread... good to keep in mind. OK, so there were several things that I tried here... I tried to patch out the checksum value to always think it was right, and that's when I found out that there were other places in the program that checksummed THIS place. It's a big mess. So basically, we know WHERE to patch (2 calls in the binary, and 2 calls in the framework), but we can't because of the checksumming. Actually, it seems like there was no checks on the main binary at all... so maybe we can leave the main binary patched and just figure out another way for the framework patches.

For those of you that want to test this theory out, put the Pro Tools framework back to the way it was (unmodified). Then start up Pro Tools under gdb. Break on 0xE8A4 (program entry point) so we can make sure that the external frameworks are all loaded. Now, set a breakpoint on the addresses we had patched in the frameworks by adding the offset to the base (run vmmap and look for the __TEXT segment for Pro Tools framework to find the base). In my case, I'm adding 0x16BB246 and 0x157E528 to 0x6DD000 to get 0x1D98246 and 0x1C5B528. After each of these breakpoints, use the "commands" feature of gdb. The command you want to run is:
return 1
c
Then continue on with gdb. Enjoy playing around with Pro Tools

EDIT: actually, breaking on 0xE8A4 may not work. Sometimes it will crash. Better option is to set "stop-on-solib-events" to 1 and keep going until you get the splash screen, and then set the breakpoints.

What's great is that at this point, with just the main binary patched, and the following gdb commands:

Code:

```
gdb$ hb *0x1D98246
Hardware assisted breakpoint 1 at 0x1d98246
gdb$ commands
>return 1
>c
>end
gdb$ hb *0x1C5B528
Hardware assisted breakpoint 2 at 0x1c5b528
gdb$ commands
>return 1
>c
>end
gdb$ c
```

you can be playing around with Pro Tools 9 RIGHT NOW!

You could even write a gdb script to perform all of this automatically. That was my first idea, until the performance hit was realized... Actually, it's only slow when loading new libraries... after having Pro Tools open for a bit, you won't even notice any slowness.

Additional Notes from 31c040c3:

Tonight I found the function responsible for checking the HD/PT2 iLok licenses and an emulator is not actually required to run the full HD or Production Toolkit 2 version. The key function is sub_16BAB82 in Pro Tools.framework. Forcing sub_16BAB82 to return 1 though will launch a fully functional Pro Tools HD. Everything else in the app shows HD and it's fully functional without any of the limits of the non-HD version.

I did some reading and learned why this was really the last missing piece of the puzzle. Previous versions of Pro Tools HD required hardware for the application to even run, but that changed with PT9 HD. While TDM (and some other features) still obviously require hardware, when the Pro Tools 9 app is launched as the HD version it checks whether or not hardware is attached, and if not it simply reverts to PT9 with Complete Production Toolkit 2 which is exactly what most people are looking for.

Saw that 9.0.1 was released and took a look at it. The main protection scheme is the same but offsets have changed. In Pro Tools.framework the offsets are now 0x157EEE2, 0x16BB542, and 0x16BBC06. In the main app they are 0xDC41, 0xF5C0, 0x48D46 and the pace loader offset is the same at 0x1BE60A.

The steps for the full crack are:

1. Install Pro Tools, along with Complete Production Toolkit 2 and HEAT from the optional installers folder
2. Use UnLOK on all necessary frameworks in **/Library/Frameworks**, **/Applications/Digidesign/Pro Tools/Pro Tools.app/Contents/Frameworks**, **/Applications/Digidesign/Pro Tools/Pro Tools Utilities/DigiTest.app/Contents/Frameworks**, and **/Applications/Digidesign/Pro Tools/SoftwareUpdater.app/Contents/Frameworks**

For the Plug-Ins folder:

Code:

```
cd /Library/Application\ Support/Digidesign/Plug-Ins
for i in *.dpm; do unpace -f "$i" -s; done
```

For the MP3 bundle (click quit when the dialog box pops up):

Code:

```
unpace -f /Applications/Digidesign/Pro\ Tools/Codecs/MP3/MP3.bundle -s
```

And after copying the UnLOKed frameworks from the Pro Tools app to minihost/Contents/Frameworks, for the Controllers folder:

Code:

```
cd /Applications/Digidesign/Pro\ Tools/Controllers
for i in *.bundle; do unpace -f "$i" -s -h minihost; done
```

3. Use the modified unpace on the app binary and manually patch the offsets **0xDC51**, **0xF5D0**, **0x48D56**, and **0x1BE60A**to return 1 with 31c040c3. This patches the pace loader and the license checks.
4. Use the original unpace on the plugins in **/Library/Application Support/Digidesign/Plug-Ins**, the bundle at: **/Applications/Digidesign/Pro Tools/Codecs/MP3/MP3.bundle**, and all bundles in **/Applications/Digidesign/Pro Tools/Controllers**. Be sure to use the -s option when using the original unpace as it had a tendency to corrupt some of the bundles when stripping them. Also, for the bundles in the Controllers folder you'll need to copy the UnLOK'd frameworks from the Pro Tools app to

minihost/Contents/Frameworks and run unpace with the
"-h minihost" option
5. For the last part, load Pro Tools into gdb and break and return on ptrace as I posted earlier in the topic (if you're not using an anti-ptrace kext). Then use "set stop-on-solib-events 1" and "continue" to load the libraries up until the splash screen, and at that point use vmmap to find out what the base address of the text segment of Pro Tools.framework is. Then take that address and set hardware breakpoints at the following points: **BASE+0x16BB246**, **BASE+0x157E528**, and**BASE+0x16BAB82**, forcing them all to return 1. These offsets are responsible for the license checks in the Pro Tools framework. Finally use "set stop-on-solib-events 0" and "continue" to load a fully licensed version of Pro Tools 9 HD.

You have to use the base+offset formula to find the breakpoints the first time, because different systems/configurations will load in the framework at different spots. For example, on my machine Pro Tools.framework always loads in at 0x903000 so my offsets are different than unsanity's previously posted ones (0x1D98246 and 0x1C5B528).You don't actually want to patch the binary in Pro Tools.framework directly, because then it will fail the checksum verifications. All the patching of Pro Tools.framework is done in memory with gdb. Regardless, a general IDA tip is that functions get their names from their offsets. For example, sub_16BAB82 in the Pro Tools framework is at offset 0x16BAB82.

General Hasp HL reversing Notes

By the_maniacz (MSJ)

From my research, I have found the following to be true in most hasp protected apps / bundle. Infact, it is very easy to reverse hasp protection in osx as long as the binary isnt encrypted. Take an OTX dump. The key protection check almost always lies within the following functions:

Code:

```
_hasp_logout:
_hasp_decrypt:
_hasp_encrypt:
_hasp_login:
_hasp_get_sessioninfo:
_hasp_update:
```

The check always occurs in the following call in the above functions:

Code:

```
calll        ___schweinshaxn02
```

Immediately after this call, there is an instruction in the following pattern:

Code:

```
beba020000          movl        $0x000002ba,%esi
89f0                movl        %esi,%eax
```

or

Code:

```
89f0                movl        %esi,%eax
```

or

Code:

```
bfbb020000          movl        $0x000002bb,%edi
89f8                movl        %edi,%eax
```

The reversing is simple. Make sure that %eax is 0 in the pattern. So, you can xorl the movl following. So for the above, the codes will be

89f0 change to 33c0
89f8 change to 33c0

Alternately, you can also change the value in esi or edi as

beba020000 to be00000000

etc.

In addition to these, there is a check within _hasp_login: as

Code:

```
_hasp_login:
+0    0008e6f2  55                pushl        %ebp
+1    0008e6f3  57                pushl        %edi
+2    0008e6f4  56                pushl        %esi
+3    0008e6f5  53                pushl        %ebx
+4    0008e6f6  81ec6c020000          subl       $0x0000026c,%esp
+10    0008e6fc  e8a4280000          calll       ___i686.get_pc_thunk.bx
+15    0008e701  8bac2480020000       movl       0x00000280(%esp),%ebp
+22    0008e708  8bbc2484020000       movl       0x00000284(%esp),%edi
+29    0008e70f  8d83112d0400          leal       0x00042d11(%ebx),%eax    e
nter hasp_login\n
+35    0008e715  890424             movl        %eax,(%esp)
+38    0008e718  e811310000          calll       ___schweinshaxn12
+43    0008e71d  89ee                movl        %ebp,%esi
+45    0008e71f  6631f6             xorw        %si,%si
+48    0008e722  81fe0000ffff          cmpl       $0xffff0000,%esi
+54    0008e728  7430                je        0x0008e75a
```

The instruction at +54 has to be changed to a jmp as

Code:

```
7430 to eb30
```

This will clear all hasp HL checks. Any additional Protection will be program protection that is not related to hasp. This method can be followed for CinemaCraft encoder and others.

Reversing Wave Rider 2

By the_maniacz (MSJ)

Hi. Today, lets look at the technique in Reversing Wave Rider 2. Now, on the website, there are 2 versions, one is the Try and the other is Buy. The note about Try said:

Quote:

> Try Wave Rider before you buy. Download the demo version that does not expire. It will give you a fully functional Wave Rider with interruptions.

That means that we can try and reverse this one.
The Description said that it will give interruptions. If you run waverider in Protools, you will see that after a certain time, the fader simply goes down and stays there for a certain time. The thing to understand is that this is usually a timer event. Before you try and reverse it, if you do a

Code:

```
grep -i -r sgemexam .
```

You will find that it is indeed Pace Protected. So, unpace as usual, and once done, pop it in OTX and get the disassemble.
Look for the word Timer or so. Keep searching and you will find:

Code:

```
CSimpleFaderMover::timerCallback(__CFRunLoopTimer*, void*)
+0   0003d254  55                    pushl       %ebp
    +1   0003d255  89e5                    movl        %esp,%ebp
    +3   0003d257  53                      pushl       %ebx
    +4   0003d258  83ec24                  subl        $0x24,%esp
    +7   0003d25b  e856580500              calll       0x00092ab6
   +12   0003d260  8b450c                  movl        0x0c(%ebp),%eax
   +15   0003d263  8945ec                  movl        %eax,0xec(%ebp)
   +18   0003d266  8b45ec                  movl        0xec(%ebp),%eax
   +21   0003d269  05bc000000              addl        $0x000000bc,%eax
   +26   0003d26e  8945f0                  movl        %eax,0xf0(%ebp)
   +29   0003d271  8b450c                  movl        0x0c(%ebp),%eax
```

```
   +32  0003d274  8945f4                  movl        %eax,0xf4(%ebp)
   +35  0003d277  837df400                cmpl        $0x00,0xf4(%ebp)
   +39  0003d27b  7432                    je          0x0003d2af

   +41  0003d27d  8b45f4                  movl        0xf4(%ebp),%eax
   +44  0003d280  0fb6809c000000          movzbl      0x0000009c(%eax),%eax
   +51  0003d287  3401                    xorb        $0x01,%al
   +53  0003d289  84c0                    testb       %al,%al
   +55  0003d28b  7422                    je          0x0003d2af

   +57  0003d28d  8b45f0                  movl        0xf0(%ebp),%eax
   +60  0003d290  f20f1008                movsd       (%eax),%xmm1
   +64  0003d294  8d83e0b40700            leal        0x0007b4e0(%ebx),%eax
   +70  0003d29a  f20f1000                movsd       (%eax),%xmm0
   +74  0003d29e  660f2ec1                ucomisd     %xmm1,%xmm0
   +78  0003d2a2  7720                    ja          0x0003d2c4

   +80  0003d2a4  8b45f4                  movl        0xf4(%ebp),%eax
   +83  0003d2a7  890424                  movl        %eax,(%esp)
   +86  0003d2aa  e83dffffff              calll       CSimpleFaderMover::move
FaderDown()

   +91  0003d2af  8b45f0                  movl        0xf0(%ebp),%eax
   +94  0003d2b2  f20f1000                movsd       (%eax),%xmm0
   +98  0003d2b6  8d83e8b40700            leal        0x0007b4e8(%ebx),%eax
  +104  0003d2bc  660f2e00                ucomisd     (%eax),%xmm0
  +108  0003d2c0  7318                    jae         0x0003d2da
        return;
```

Notice there is a call called **CSimpleFaderMover::moveFaderDown()** at 0x0003d2aa. Before that there is a

Code:

```
0003d2a2  7720                  ja          0x0003d2c4
```

If that were to jump, then it would go past the **CSimpleFaderMover::moveFaderDown()**.
So lets change that to a jmp as

Code:

```
7720 to EB20
```

Save and Run. You will see that Wave Rider works flawlessly!!
There is an additional change that you need to do for this to be a full reversal. Although it now is fully functional, you will see that it says DEMO in the plugin window.
If you inspect the Resource file, you will see that it contains the background for the fully functional as well as the Demo. If you interchange their ID (in Resourcer or some similar resource editor), you will have a fully working Plugin.
This is just posted so that the developer can understand that by having a simple check, the full program itself is given away.